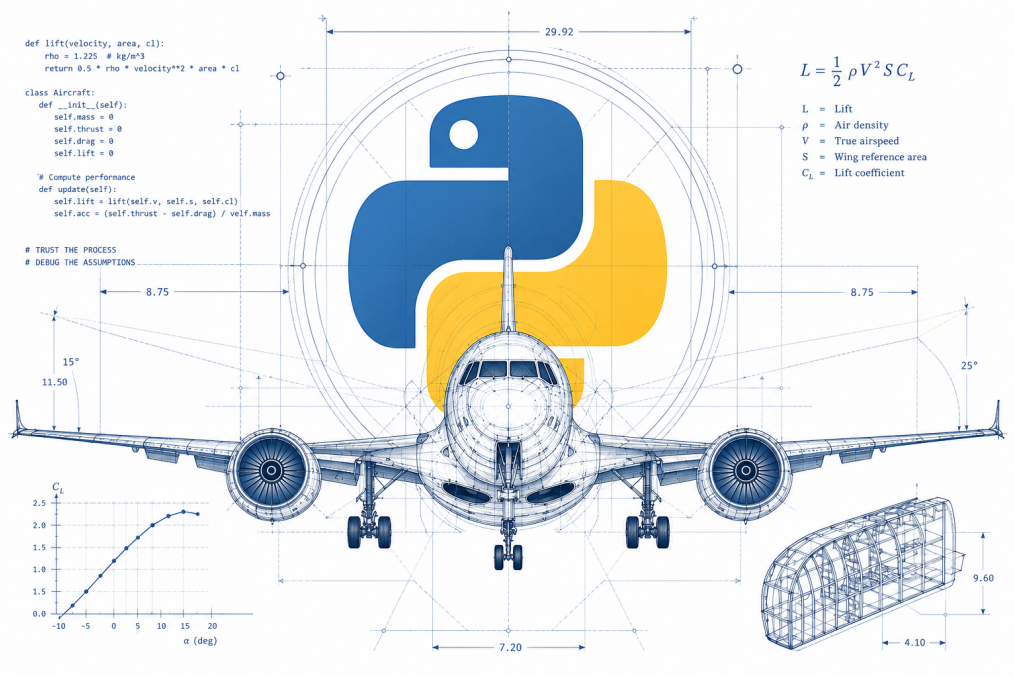
Python per l'Aeronautica

Monografia operativa di programmazione
per il corso di Costruzioni Aeronautiche



Prof. Ing. Biagio Raucci

Istituto Tecnico – Indirizzo Trasporti e Logistica
Articolazione Costruzione del Mezzo Aereo

Collana didattica SCSI • `braucci.github.io/SCSI/`

Anno scolastico 2026–2027

A G.,

al quale avrei voluto insegnare, una per una,
tutte le cose scritte in queste pagine.

Nella speranza che, leggendole, gli giunga almeno
intera la misura della stima che ho
per la sua intelligenza.

«Non dar retta ai tuoi occhi, e non credere a quello che vedi. Gli occhi vedono solo ciò che è limitato. Guarda col tuo intelletto, e scopri quello che conosci già, allora imparerai come si vola.»

Richard Bach
Il gabbiano Jonathan Livingston

Premessa

Questa monografia nasce da una constatazione semplice: nel corso di Costruzioni Aeronautiche i calcoli ripetitivi — le tabelle dell'atmosfera standard, i punti della polare, le velocità caratteristiche, i diagrammi delle sollecitazioni lungo l'ala — sono il terreno naturale in cui un linguaggio di programmazione smette di essere una materia astratta e diventa uno *strumento di officina*, al pari della calcolatrice e del regolo di un tempo.

Il volume è pensato per studenti che *non hanno mai programmato*. Per questo la teoria è ridotta al minimo indispensabile: si introduce un concetto solo quando serve a risolvere un problema del corso, e ogni concetto è immediatamente messo alla prova su un caso aeronautico concreto. Non troverete esercizi sui numeri primi o sulle anagrafiche: troverete la troposfera, il tubo di Pitot, il longherone alare.

Come è organizzato il volume

La **Parte I** (*Fondamenti per tutti*) è comune alle tre classi: nei capitoli iniziali fornisce l'intero bagaglio linguistico necessario — variabili, decisioni, cicli, funzioni, liste, NumPy e Matplotlib — più un capitolo di metodo sull'uso dell'intelligenza artificiale come assistente di programmazione. La chiude un capitolo propedeutico dedicato al **calcolo e alla visualizzazione dei vettori**, che raccorda questo volume alla monografia *Teoria dei vettori* della stessa collana: ogni programma vi è collaudato su un esercizio svolto di quel testo, di cui riproduce il risultato cifra per cifra. È il capitolo da cui conviene passare prima di affrontare qualunque parte annuale, perché forze, velocità e momenti — cioè quasi tutto ciò che segue — sono vettori.

Le **Parti II, III e IV** sono invece *annuali*: ciascuna raccoglie **dieci problemi risolti al calcolatore** — trenta in tutto — agganciati agli argomenti che la classe sta studiando in teoria, secondo la scansione del corso:

- **terzo anno**: unità aeronautiche, atmosfera standard fino alla stratosfera, aerostatica, vettori, anemometria, Reynolds e Mach, geometria dell'ala e dei profili NACA;
- **quarto anno**: polare del velivolo, volo livellato, salita, virata, planata, decollo, crociera, autonomie, inviluppo di manovra e di raffica;
- **quinto anno**: trave e longherone, geometria delle sezioni, flussi di taglio, torsione di Bredt, giunzioni rivettate, aste di controventatura, fusoliera pressurizzata, flusso comprimibile, preparazione alla seconda prova.

Ogni scheda ha la stessa struttura: *problema*, *richiami teorici minimi* (la formula chiave, con la dimostrazione dove è breve e istruttiva), *prompt pronto all'uso* costruito con le cinque componenti del capitolo di metodo — perché il prompt è l'atto di progetto, e va studiato quanto il codice —, *programma commentato riga per riga*, *output verificato*, *validazione manuale* di almeno un valore per una via indipendente, ed *esercizi proposti*.

Chiude il volume la **Parte V**, un *corso avanzato* di introduzione alla fluidodinamica computazionale rivolto a chi proseguirà gli studi in ingegneria: dodici esercitazioni in progressione, dal trasporto di un impulso alle equazioni di Navier–Stokes risolte su due problemi canonici, con la stessa disciplina di validazione del resto del volume — quasi ogni esercitazione si confronta con una soluzione esatta.

Il patto d'aula: l'IA scrive, tu rispondi

In questo corso l'uso dell'intelligenza artificiale non è vietato: è *disciplinato*. L'IA può generare codice più in fretta di chiunque, ma la responsabilità del risultato — come per ogni calcolo di verifica strutturale — resta dell'ingegnere. Il patto è dunque questo: potete far scrivere il codice all'IA, ma dovete essere in grado di *leggerlo*,

spiegarlo e soprattutto *validarlo* contro un calcolo manuale o un valore tabellato. Un programma che fornisce $\rho = 0{,}736\ \mathrm{kg/m^3}$ a 5000 m è accettabile solo se sapete verificare quel numero sulla tabella ICAO. Il capitolo 7 formalizza questo metodo.

Regola aurea della collana

Tutti i valori numerici stampati in questo volume sono stati ottenuti *eseguendo realmente* i programmi riportati: nessun output è trascritto a mano. È la stessa disciplina che chiediamo a voi.

B. Raucci

Indice

Parte I

Fondamenti per tutti

Il linguaggio essenziale, dal primo comando al primo grafico

Capitolo 1

L'officina digitale: dove si scrive il codice

In questo capitolo imparerai a

- distinguere *editor*, *interprete* e *script*;
- eseguire il primo programma Python in tre ambienti diversi (Thonny, Google Colab, terminale);
- leggere un messaggio d'errore senza spaventarti.

1.1 Che cosa significa "programmare"

Un programma è una *procedura di calcolo scritta in un linguaggio non ambiguo*. In questo non è diverso da una check-list di cabina o da una relazione di calcolo strutturale: una sequenza ordinata di operazioni che chiunque — anche una macchina — può eseguire ottenendo lo stesso risultato. Python è il linguaggio che useremo perché è *interpretato* (si esegue riga per riga, senza compilazione), ha una sintassi leggibile quasi come l'italiano tecnico ed è lo standard di fatto nel calcolo scientifico e in industria aerospaziale (NASA, ESA e i costruttori lo usano quotidianamente per pre e post–processing).

Definizione

L'**interprete** Python è il programma che legge il vostro file sorgente (lo *script*, un file di testo con estensione `.py`) e ne esegue le istruzioni una alla volta, dall'alto verso il basso.

1.2 Tre modi per lavorare

1.2.1 Thonny (in laboratorio)

Thonny (`https://thonny.org`) è l'ambiente che useremo in laboratorio: leggero, gratuito, pensato per la didattica. All'apertura mostra due aree: l'*editor* in alto (dove si scrive lo script) e la *shell* in basso (dove compare l'output). Si esegue con il tasto verde o con `F5`.

1.2.2 Google Colab (a casa, senza installare nulla)

Colab (`https://colab.research.google.com`) esegue Python nel browser: basta un account scolastico. È l'ambiente ideale per i compiti a casa perché non richiede installazioni e i *notebook* si condividono con un link, come i documenti.

1.2.3 Il terminale (per capire cosa accade davvero)

Uno script si può eseguire anche direttamente:

```
Prompt dei comandi

C:\Users\studente> python primo.py
```

1.3 Il primo programma

Aprite l'editor, scrivete la riga seguente e salvate come `primo.py`:

primo.py

```
print("ITIS Majorana - corso di Costruzioni Aeronautiche")
```

Output

```
ITIS Majorana - corso di Costruzioni Aeronautiche
```

`print(...)` è una *funzione*: riceve fra parentesi un dato (qui una *stringa* di testo, delimitata da virgolette) e lo scrive a schermo. Tutto il corso, in fondo, sarà una variazione su questo tema: dare dati in ingresso, elaborare, stampare risultati — esattamente ciò che fa una relazione di calcolo.

Attenzione

Python distingue le maiuscole: `print` funziona, `Print` no. E gli errori di battitura non "più o meno funzionano": o il programma è sintatticamente corretto, o l'interprete si ferma. Questa severità è un pregio: è la stessa che pretendiamo da un disegno costruttivo.

1.4 Leggere gli errori: il collaudo comincia qui

Provate deliberatamente a sbagliare:

errore.py

```
print("Manca la parentesi di chiusura"
```

Output

```
File "errore.py", line 1
    print("Manca la parentesi di chiusura"
          ^
SyntaxError: '(' was never closed
```

Il messaggio dice *tutto*: il file, la riga (`line 1`), il punto (`^`) e la diagnosi (parentesi mai chiusa). Un messaggio d'errore non è un rimprovero: è il verbale di collaudo che indica esattamente dove intervenire. Imparare a leggerlo è la prima competenza professionale del corso.

Dialoga con l'IA

Quando un errore non vi è chiaro, incollate all'IA *il messaggio completo* insieme al codice e chiedete: *"Spiegami questo errore riga per riga e dimmi come correggerlo, senza riscrivere tutto il programma"*. Pretendete la spiegazione, non solo la correzione: al colloquio orale dovrete saper spiegare voi.

Prova tu

1. Scrivete un programma che stampi su tre righe: il vostro nome, la classe, il velivolo che preferite.
2. Provocate volontariamente tre errori diversi (virgolette mancanti, parentesi, nome `prin`) e trascrivete sul quaderno la diagnosi che l'interprete fornisce per ciascuno.

Capitolo 2

Variabili e calcoli: la pressione dinamica

In questo capitolo imparerai a

- usare variabili e operatori aritmetici;
- distinguere i tipi `int`, `float`, `str`;
- leggere dati con `input()` e formattare l'output con le *f-string*;
- calcolare pressione dinamica e portanza di un velivolo.

2.1 Le variabili: la nomenclatura del calcolo

In una relazione di calcolo scriviamo "sia $\rho = 1{,}225$ kg/m^3 la densità dell'aria". In Python la stessa dichiarazione è un'*assegnazione*:

Codice Python

```
rho = 1.225
```

Il simbolo = non è l'uguaglianza matematica: si legge *"assegna a"*. Una variabile è un'etichetta attaccata a un valore in memoria; il valore può cambiare durante l'esecuzione (per questo si chiama *variabile*), esattamente come ρ cambia con la quota.

Osservazione

Scegliete nomi parlanti e coerenti con la nomenclatura del corso: `rho`, `V`, `S`, `CL`, `q`. Un programma con variabili `a`, `b`, `x1` è illeggibile quanto una relazione senza simbologia: fra sei mesi non lo capireste nemmeno voi.

2.2 I tipi di dato essenziali

Python assegna da solo il *tipo* in base al valore (tipizzazione dinamica). A noi ne servono tre:

Tipo	Significato	Esempio aeronautico
`int`	numero intero	`n_rivetti = 6`
`float`	numero con la virgola	`rho = 1.225`
`str`	stringa di testo	`profilo = "NACA 2412"`

La funzione `type(x)` rivela il tipo di una variabile; le funzioni `int()`, `float()`, `str()` convertono da un tipo all'altro.

Attenzione

In Python il separatore decimale è il *punto*, non la virgola: `1.225`, mai `1,225`. La virgola in Python separa più valori (lo vedremo con le liste), quindi `1,225` sarebbe interpretato come la coppia (1; 225): nessun

errore a schermo, ma risultati privi di senso. È l'errore più insidioso per chi è abituato alla convenzione italiana.

2.3 Gli operatori

`+`, `-`, `*`, `/` funzionano come vi aspettate; `**` è l'elevamento a potenza (`V**2` è V^2); `//` è la divisione intera e `%` il resto. La precedenza è quella algebrica usuale e le parentesi tonde raggruppano.

2.4 Il primo calcolo aeronautico: $q = \frac{1}{2}\rho V^2$

Formula chiave

La **pressione dinamica** è l'energia cinetica per unità di volume della corrente:

$$q = \tfrac{1}{2}\rho V^2 \qquad [\text{Pa}]$$

È la grandezza di riferimento di *tutta* l'aerodinamica: portanza e resistenza si scrivono $L = q\,S\,C_L$ e $D = q\,S\,C_D$.

`pressione_dinamica.py`

```
# pressione_dinamica.py
# Calcolo della pressione dinamica q = 1/2 * rho * V^2
rho = 1.225          # densita' dell'aria al livello del mare [kg/m^3]
V = 50.0             # velocita' di volo [m/s]
q = 0.5 * rho * V**2
print("Pressione dinamica q =", q, "Pa")
```

Output

```
Pressione dinamica q = 1531.25 Pa
```

Osservazione

Tutto ciò che segue `#` è un *commento*: l'interprete lo ignora, ma per chi legge è prezioso quanto le note a margine di un disegno. In questo corso ogni variabile fisica va commentata con significato e *unità di misura*: un programma senza unità è inaffidabile quanto un calcolo senza unità.

Validazione manuale. $q = 0{,}5 \cdot 1{,}225 \cdot 50^2 = 0{,}5 \cdot 1{,}225 \cdot 2500 = 1531{,}25$ Pa. Il programma è verificato.

2.5 Input dall'utente e output formattato

`input("...")` sospende il programma, mostra il messaggio e restituisce ciò che l'utente digita — sempre come *stringa*: per usarlo nei calcoli va convertito con `float()`. Per l'output, le *f-string* permettono di inserire variabili (e formati) dentro il testo: `f"q = {q:.1f} Pa"` stampa q con una cifra decimale.

portanza.py

```
rho = 1.225
S = 16.2
CL = 0.62
V = float(input("Velocita' di volo [m/s]: "))
q = 0.5 * rho * V**2
L = q * S * CL
print(f"q = {q:.1f} Pa")
print(f"Portanza L = {L:.0f} N (equivalente a {L/9.80665:.0f} kg)")
```

Output

```
Velocita' di volo [m/s]: 50
q = 1531.2 Pa
Portanza L = 15380 N (equivalente a 1568 kg)
```

Con $S = 16{,}2\ \text{m}^2$ e $C_L = 0{,}62$ (valori tipici di un monomotore da addestramento) la portanza a 50 m/s vale 15 380 N: il velivolo può sostenere circa 1568 kg. Ecco perché un aereo di 1100 kg a quella velocità *sale*: la portanza disponibile eccede il peso.

Dialoga con l'IA

Chiedete all'IA: *"Modifica `portanza.py` perché chieda anche la quota e usi la densità ISA corrispondente"*. Poi *verificate* il codice ricevuto: usa la formula della troposfera corretta? La densità a 5000 m risulta $\approx 0{,}736\ \text{kg/m}^3$? Se non sapete rispondere, non consegnate.

Prova tu

1. Convertitore di velocità: chiede i nodi e stampa m/s e km/h (ricordate: $1\,\text{kt} = 1852/3600 = 0{,}5144\ \text{m/s}$).
2. Calcolate la resistenza $D = q\,S\,C_D$ con $C_D = 0{,}032$ e gli stessi dati dell'esempio; verificate a mano il risultato.
3. Un tubo di Pitot misura $\Delta p = 1531\ \text{Pa}$: ricavate $V = \sqrt{2\Delta p/\rho}$ (serve `import math` e `math.sqrt`, anticipato dal capitolo 4).

Capitolo 3

Decisioni e ripetizioni: dal regime di Mach alla tabella ISA

In questo capitolo imparerai a

- far prendere decisioni al programma con `if/elif/else`;
- ripetere calcoli con `while` e `for`;
- classificare il regime di volo e generare la tabella della troposfera.

3.1 Le decisioni: `if`, `elif`, `else`

Molte regole del corso sono *condizionali*: "*se* $M < 0{,}8$ il regime è subsonico"; "*se* $n > n_{max}$ la manovra è fuori inviluppo". Python le esprime così:

regime_mach.py

```
M = float(input("Numero di Mach: "))
if M < 0.8:
    regime = "subsonico"
elif M < 1.2:
    regime = "transonico"
elif M < 5.0:
    regime = "supersonico"
else:
    regime = "ipersonico"
print(f"M = {M}: regime {regime}")
```

Output

```
Numero di Mach: 0.65
M = 0.65: regime subsonico
```

Output

```
Numero di Mach: 2.3
M = 2.3: regime supersonico
```

Due osservazioni fondamentali.

1. **L'indentazione è sintassi.** Le righe rientrate di 4 spazi *appartengono* al blocco dell'`if`. In altri linguaggi i blocchi si delimitano con parentesi graffe; Python usa il rientro, e sbagliare il rientro cambia il significato del programma.

2. **Le condizioni si valutano in ordine.** Arrivati a `elif M < 1.2` sappiamo già che $M \geq 0{,}8$: la seconda condizione sottintende la prima. Se invertiste l'ordine dei rami, un $M = 0{,}65$ verrebbe classificato transonico. La logica del programma va *dimostrata*, non sperata.

Gli operatori di confronto sono `<`, `<=`, `>`, `>=`, `==` (uguaglianza: *doppio* uguale!) e `!=` (diverso); le condizioni si combinano con `and`, `or`, `not`.

3.2 Il ciclo `while`: la tabella della troposfera

Nel modello ISA la temperatura decresce linearmente con la quota, $T(h) = T_0 - \lambda h$ con $\lambda = 6{,}5 \times 10^{-3}$ K/m, fino alla tropopausa (11 000 m). Compilare la tabella a mano ogni 1000 m è il classico lavoro da delegare a un ciclo:

tabella_isa.py

```
T0 = 288.15       # [K]
lam = 0.0065      # gradiente termico verticale [K/m]
h = 0
while h <= 11000:
    T = T0 - lam * h
    print(f"h = {h:6d} m   T = {T:7.2f} K = {T-273.15:7.2f} gradi C")
    h = h + 1000
```

Output

```
h =      0 m   T =  288.15 K =   15.00 gradi C
h =   1000 m   T =  281.65 K =    8.50 gradi C
h =   2000 m   T =  275.15 K =    2.00 gradi C
h =   3000 m   T =  268.65 K =   -4.50 gradi C
h =   4000 m   T =  262.15 K =  -11.00 gradi C
h =   5000 m   T =  255.65 K =  -17.50 gradi C
h =   6000 m   T =  249.15 K =  -24.00 gradi C
h =   7000 m   T =  242.65 K =  -30.50 gradi C
h =   8000 m   T =  236.15 K =  -37.00 gradi C
h =   9000 m   T =  229.65 K =  -43.50 gradi C
h =  10000 m   T =  223.15 K =  -50.00 gradi C
h =  11000 m   T =  216.65 K =  -56.50 gradi C
```

Validazione. A 11 000 m: $T = 288{,}15 - 0{,}0065 \cdot 11000 = 288{,}15 - 71{,}5 = 216{,}65$ K $= -56{,}5\,^{\circ}$C, che è esattamente il valore ICAO della tropopausa. Verificato.

Attenzione

Un `while` la cui condizione non diventa mai falsa è un *ciclo infinito*: il programma non termina. Se dimenticate la riga `h = h + 1000`, la quota resta 0 per sempre. Quando accade, fermate l'esecuzione con `Ctrl+C` (o il tasto stop di Thonny) e chiedetevi: *quale istruzione fa progredire il ciclo verso la sua condizione d'uscita?*

3.3 Il ciclo `for`: iterare su una sequenza nota

Quando le iterazioni sono note in anticipo, `for` è più naturale: `for h in range(0, 12000, 1000):` percorre le quote da 0 a 11 000 a passi di 1000 (l'estremo destro di `range` è *escluso*: dettaglio da ricordare). Dentro un ciclo, `break` esce immediatamente, `continue` salta all'iterazione successiva.

Esempio – quota di tangenza teorica di un pallone

Un ciclo `for` con `break` trova la prima quota alla quale la densità ISA scende sotto un valore assegnato: si percorre la tabella e ci si ferma appena la condizione è soddisfatta. Riprenderemo l'idea nella scheda di aerostatica del terzo anno.

Dialoga con l'IA

Fate generare all'IA un programma che classifichi il regime con soglie sbagliate (ad esempio dite: "il transonico va da 0,9 a 1,5") e poi *correggetelo voi* riportando le soglie del corso (0,8–1,2). Esercizio speculare al solito: qui l'errore lo introduce l'IA su vostra istruzione, e il collaudatore siete voi.

Prova tu

1. Estendete `regime_mach.py` perché rifiuti con un messaggio i Mach negativi.
2. Riscrivete `tabella_isa.py` con un `for` e `range`.
3. Tabella dei fattori di carico in virata corretta: $n = 1/\cos\phi$ per ϕ da $0°$ a $75°$ a passi di $15°$ (usate `math.cos` e `math.radians`). A che angolo si supera $n = 2$?

Capitolo 4

Funzioni e liste: costruiamo `atmosfera.py`

In questo capitolo imparerai a

- definire funzioni con `def`, parametri e `return`;
- importare moduli (`math`) e riusare il proprio codice;
- usare le liste per gestire serie di dati sperimentali;
- scrivere la libreria `atmosfera.py` che vi accompagnerà per tre anni.

4.1 Le funzioni: incapsulare una formula

Una funzione è una formula messa in scatola: riceve i dati in ingresso (i *parametri*), esegue il calcolo, *restituisce* il risultato con `return`. È l'equivalente informatico della formula incorniciata di una relazione: la si dimostra una volta e la si *richiama* ogni volta che serve.

Formula chiave

Nella troposfera ($0 \le h \le 11\,000$ m) il modello ISA fornisce:

$$T = T_0 - \lambda h, \qquad p = p_0 \left(\frac{T}{T_0}\right)^{\frac{g}{R\lambda}}, \qquad \rho = \frac{p}{R\,T}$$

con $T_0 = 288{,}15$ K, $p_0 = 101\,325$ Pa, $\lambda = 6{,}5 \times 10^{-3}$ K/m, $g = 9{,}806\,65$ m/s^2, $R = 287{,}05$ J/(kg K). L'esponente vale $g/(R\lambda) = 9{,}80665/(287{,}05 \cdot 0{,}0065) \simeq 5{,}256$.

`atmosfera.py`

```
# atmosfera.py -- modello ISA della troposfera (0-11000 m)
import math

T0   = 288.15      # temperatura al livello del mare [K]
p0   = 101325.0    # pressione al livello del mare [Pa]
lam  = 0.0065      # gradiente termico verticale [K/m]
g    = 9.80665     # accelerazione di gravita' [m/s^2]
R    = 287.05      # costante dell'aria [J/(kg K)]

def isa(h):
    """Restituisce (T, p, rho) alla quota h [m] nella troposfera."""
    T = T0 - lam * h
    p = p0 * (T / T0) ** (g / (R * lam))
    rho = p / (R * T)
    return T, p, rho

if __name__ == "__main__":
    for h in [0, 2000, 5000, 8000, 11000]:
        T, p, rho = isa(h)
```

```
        print(f"h = {h:6d} m   T = {T:7.2f} K   "
              f"p = {p:9.1f} Pa   rho = {rho:6.4f} kg/m^3")
```

Output

```
h =      0 m   T =  288.15 K   p =  101325.0 Pa   rho = 1.2250 kg/m^3
h =   2000 m   T =  275.15 K   p =   79495.0 Pa   rho = 1.0065 kg/m^3
h =   5000 m   T =  255.65 K   p =   54019.5 Pa   rho = 0.7361 kg/m^3
h =   8000 m   T =  236.15 K   p =   35599.4 Pa   rho = 0.5252 kg/m^3
h =  11000 m   T =  216.65 K   p =   22631.7 Pa   rho = 0.3639 kg/m^3
```

Validazione. Confrontiamo con la tabella ICAO: a 2000 m la manualistica dà $p = 79\,495$ Pa e $\rho = 1{,}0066$ kg/m^3; a 11 000 m, $p = 22\,632$ Pa. Le piccolissime differenze all'ultima cifra dipendono dal valore di R adottato: sono ordini di grandezza inferiori a ogni tolleranza ingegneristica.

Tre novità da fissare:

- `import math` carica il *modulo* matematico (`math.sqrt`, `math.pi`, `math.cos`, `math.atan`…);
- la stringa tra triple virgolette subito dopo `def` è la *docstring*: la targhetta della funzione, che ne dichiara ingressi e uscite;
- la funzione restituisce *tre* valori insieme (una *tupla*); chi la chiama li "spacchetta" con `T, p, rho = isa(h)`.

Osservazione

La riga `if __name__ == "__main__":` separa il collaudo dalla libreria: il blocco si esegue solo lanciando il file direttamente. Da un altro programma potrete invece scrivere `from atmosfera import isa` e riusare la funzione: il file diventa la *vostra prima libreria*, e la userete in quasi tutte le schede delle parti annuali.

4.2 Le liste: serie di dati

Una lista è una sequenza ordinata di valori fra parentesi quadre. È lo strumento naturale per i dati di una prova: quote rilevate, tempi, letture di galleria.

salita_liste.py

```
quote = [0, 500, 1200, 2100, 3300]        # quote raggiunte [m]
tempi = [0, 2.1, 5.0, 9.2, 15.5]          # tempi corrispondenti [min]

print("Numero di rilevamenti:", len(quote))
print("Quota massima:", max(quote), "m")

# velocita' media di salita in ciascun tratto
for i in range(1, len(quote)):
    dh = quote[i] - quote[i-1]            # [m]
    dt = (tempi[i] - tempi[i-1]) * 60     # [s]
    print(f"Tratto {i}: rateo medio = {dh/dt:5.2f} m/s")
```

Output

```
Numero di rilevamenti: 5
Quota massima: 3300 m
Tratto 1: rateo medio =  3.97 m/s
Tratto 2: rateo medio =  4.02 m/s
Tratto 3: rateo medio =  3.57 m/s
Tratto 4: rateo medio =  3.17 m/s
```

Si accede agli elementi con l'indice fra quadre, *contando da zero*: `quote[0]` è il primo elemento, `quote[-1]` l'ultimo. Il rateo che decresce con la quota non è un caso: la densità cala, l'eccesso di potenza pure — la fisica affiora già dai dati.

Nota

Esistono anche le *list comprehension*, scorciatoie eleganti per generare liste: `[0.5*1.225*v**2 for v in range(10, 60, 10)]` produce le pressioni dinamiche alle velocità da 10 a 50 m/s. Usatele quando la leggibilità ne guadagna, non per esibizione.

Dialoga con l'IA

Chiedete all'IA di aggiungere ad `atmosfera.py` il tratto stratosferico isotermo (da 11 000 m–20 000 m, dove $p = p_{11}\, e^{-g(h-h_{11})/(R\,T_{11})}$) e *validate* il risultato: alla tangenza di un liner, 12 000 m, deve risultare $p \approx 19\,330$ Pa. Se il valore non torna, individuate voi la riga sbagliata prima di chiedere la correzione.

Prova tu

1. Scrivete `def q_dinamica(rho, V):` e usatela per rifare l'esercizio del tubo di Pitot del capitolo precedente.
2. Scrivete `def virata(V, phi):` che restituisca fattore di carico $n = 1/\cos\phi$ e raggio $R = V^2/(g\tan\phi)$.
3. Data la lista dei C_L misurati in galleria `[0.21, 0.44, 0.67, 0.89, 1.10]` a incidenze $\alpha = 0, 2, 4, 6, 8$ gradi, calcolate la pendenza media $\Delta C_L/\Delta\alpha$ per grado e confrontatela con il valore teorico $2\pi^2/180 \simeq 0{,}11$ del profilo sottile.

Capitolo 5

NumPy e Matplotlib: calcolare e disegnare come in ufficio tecnico

In questo capitolo imparerai a

- operare su interi vettori di dati con NumPy;
- tracciare grafici tecnici con Matplotlib;
- costruire la polare del velivolo e leggerne l'efficienza.

5.1 NumPy: la fine dei cicli sui numeri

Con le liste, per calcolare $C_D = C_{D0} + k\,C_L^2$ su venti valori di C_L serve un ciclo. NumPy introduce l'`array`, un vettore su cui le operazioni agiscono *elemento per elemento*, come in algebra: la formula si scrive una volta sola e vale per tutto il vettore. È la stessa economia di scrittura del calcolo matriciale.

Formula chiave

La **polare parabolica** del velivolo completo:

$$C_D = C_{D0} + k\,C_L^2, \qquad k = \frac{1}{\pi\, e\, \mathcal{A}}$$

lega la resistenza alla portanza. L'efficienza $E = C_L/C_D$ è massima per $C_L^* = \sqrt{C_{D0}/k}$ e vale $E_{max} = 1/\left(2\sqrt{C_{D0}\,k}\right)$.

`polare_numpy.py`

```
import numpy as np

CD0 = 0.025          # coefficiente di resistenza parassita
k   = 0.045          # fattore di resistenza indotta

CL = np.arange(0.0, 1.61, 0.2)       # vettore dei CL
CD = CD0 + k * CL**2                 # NumPy opera su TUTTO il vettore
E  = np.divide(CL, CD, out=np.zeros_like(CL), where=CD>0)

print(" CL      CD       E")
for cl, cd, e in zip(CL, CD, E):
    print(f"{cl:4.2f}  {cd:7.4f}  {e:6.2f}")

print(f"\nEfficienza massima E = {E.max():.2f} per CL = {CL[E.argmax()]:.2f}")
```

```
Output

CL      CD       E
0.00   0.0250    0.00
0.20   0.0268    7.46
0.40   0.0322   12.42
0.60   0.0412   14.56
0.80   0.0538   14.87
1.00   0.0700   14.29
1.20   0.0898   13.36
1.40   0.1132   12.37
1.60   0.1402   11.41

Efficienza massima E = 14.87 per CL = 0.80
```

Validazione — e una lezione sulla discretizzazione. La teoria dà $C_L^* = \sqrt{0{,}025/0{,}045} = 0{,}745$ e $E_{max} = 1/(2\sqrt{0{,}025 \cdot 0{,}045}) = 14{,}91$. Il programma trova invece 14,87 in $C_L = 0{,}80$: perché? Perché il vettore campiona i C_L a passi di 0,2 e il massimo vero, 0,745, *cade fra due punti della griglia*. Infittendo il passo (`np.arange(0, 1.61, 0.001)`) il massimo numerico converge a quello analitico. Ricordatelo sempre: un calcolo discreto approssima il continuo, e la finezza della griglia è una scelta ingegneristica.

5.2 Matplotlib: il grafico tecnico

grafico_polare.py

```python
import numpy as np
import matplotlib.pyplot as plt

CD0, k = 0.025, 0.045
CL = np.linspace(0, 1.6, 200)        # 200 punti equispaziati
CD = CD0 + k * CL**2

plt.plot(CD, CL)                     # per convenzione: CD in ascissa
plt.xlabel("$C_D$")
plt.ylabel("$C_L$")
plt.title("Polare del velivolo")
plt.grid(True)
plt.savefig("polare.png", dpi=150)   # per la relazione
plt.show()
```

Osservazione

Notare la convenzione del corso: nella polare C_D sta in *ascissa* e C_L in *ordinata*, così la pendenza della retta per l'origine è proprio l'efficienza E, e E_{max} corrisponde alla tangente condotta dall'origine. Un grafico tecnico senza etichette degli assi e unità è da respingere quanto un disegno senza quote: `xlabel`, `ylabel`, `title` e `grid` non sono ornamenti.

Dialoga con l'IA

Chiedete: *"Aggiungi al grafico della polare il punto di efficienza massima calcolato analiticamente e la retta per l'origine tangente alla polare"*. Verificate che il punto tracciato sia (0,050; 0,745): se l'IA lo

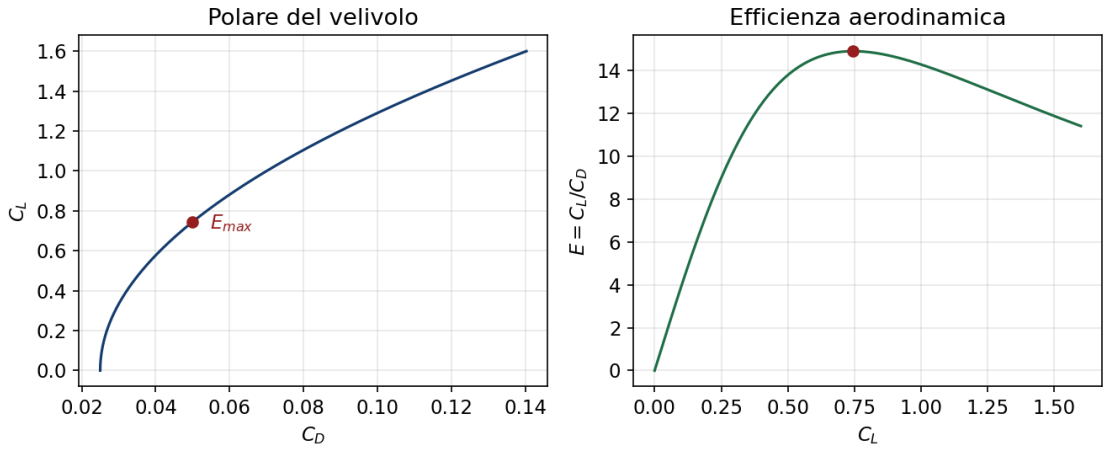


Figura 5.1. La polare C_L–C_D e la curva di efficienza prodotte con Matplotlib. Il punto rosso è la condizione di E_{max}: geometricamente, il punto di tangenza della retta uscente dall'origine.

mette altrove, la tangente non toccherà la curva — e il grafico stesso vi farà da collaudo.

Prova tu

1. Tracciate $T(h)$, $p(h)$, $\rho(h)$ della troposfera in tre sottografici affiancati (cercate `plt.subplots`), importando la *vostra* `atmosfera.py`.
2. Infittite la griglia della polare fino a ottenere $E_{max} = 14{,}91$ con due decimali esatti: quale passo è stato necessario?
3. Sovrapponete due polari con $C_{D0} = 0{,}025$ e $C_{D0} = 0{,}035$ (carrello estratto) e commentate l'effetto su E_{max}.

Capitolo 6

La grafica tecnica: disegnare per capire

In questo capitolo imparerai a

- conoscere l'anatomia di una figura Matplotlib e i suoi elementi obbligatori;
- confrontare famiglie di curve al variare di un parametro;
- comporre pannelli di sottografici, doppi assi e scale logaritmiche;
- annotare un grafico fino a renderlo pronto per una relazione tecnica;
- conoscere le regole di qualità (e gli errori tipici) del disegno dei dati.

Nota

Il capitolo 5 ha introdotto Matplotlib quanto basta per tracciare una curva. Questo capitolo la porta al livello che serve davvero a un tecnico: perché in ingegneria il grafico non è la decorazione finale di un calcolo — è uno *strumento di comprensione e di validazione*. Una tabella di venti numeri si legge con fatica; la stessa tabella disegnata rivela in un colpo d'occhio il minimo, la tendenza, l'anomalia. Molte delle relazioni che studierete (la curva di potenza, la polare, l'inviluppo di volo) *sono* grafici prima ancora che formule: saperli produrre bene è parte del mestiere quanto saperli leggere.

6.1 L'anatomia di una figura

Matplotlib organizza ogni disegno in una gerarchia di tre livelli: la *figura* (`fig`, il foglio), uno o più *assi* (`ax`, il riquadro con le scale, dove si disegna) e gli *artisti* (curve, testi, frecce). La riga che apre quasi ogni programma grafico,

```
fig, ax = plt.subplots(figsize=(6.4, 4.0))
```

crea foglio e riquadro insieme; da lì in poi si lavora *sui metodi di* `ax`. Questa impostazione «orientata agli oggetti» è quella da adottare fin da subito: rende naturale tutto ciò che segue (pannelli, doppi assi) e rispecchia il modo in cui pensiamo un disegno tecnico — un contenitore, delle scale, dei tratti.

Una figura tecnica *completa* ha sei elementi obbligatori, che la figura 6.1 mostra su un esempio reale: titolo, etichette degli assi *con simbolo e unità di misura*, griglia leggera, legenda quando le curve sono più d'una, ed eventuali annotazioni sui punti notevoli. Un grafico senza unità sugli assi è, per un ingegnere, un grafico senza significato: è la prima voce della lista di controllo di fine capitolo.

`anatomia.py -- la struttura da cui partire sempre`

```
# anatomia.py -- la figura tecnica completa dei suoi sei elementi
import numpy as np
import matplotlib.pyplot as plt

W, S, rho = 1100*9.80665, 16.2, 1.225
CD0, k = 0.025, 0.045

V  = np.linspace(26.5, 75, 300)
CL = 2*W/(rho*S*V**2)
```

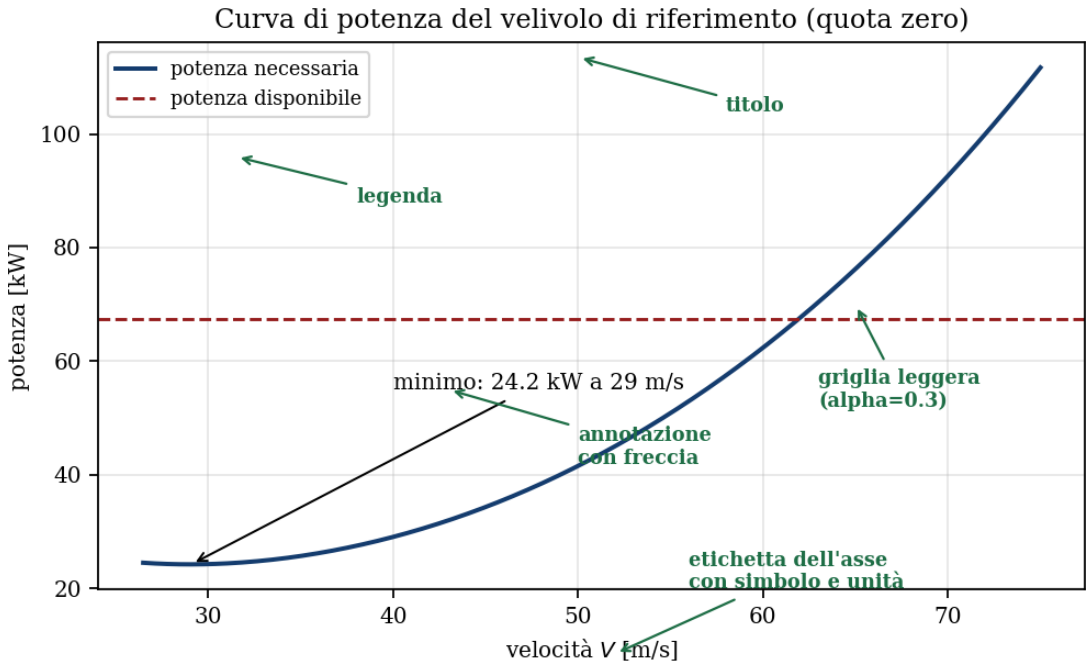


Figura 6.1. L'anatomia di una figura tecnica sui dati veri del velivolo di riferimento: ogni elemento indicato in verde è prodotto da una riga di codice precisa — titolo, legenda, griglia, etichette con unità, annotazione con freccia.

```
Pn = 0.5*rho*V**3*S*(CD0 + k*CL**2)/1000        # [kW]

fig, ax = plt.subplots(figsize=(6.4, 4.0))
ax.plot(V, Pn, lw=1.8, label="potenza necessaria")
ax.axhline(67.5, ls="--", color="firebrick", label="potenza disponibile")

i = Pn.argmin()                                  # il punto notevole
ax.annotate(f"minimo: {Pn[i]:.1f} kW a {V[i]:.0f} m/s",
            xy=(V[i], Pn[i]), xytext=(40, 55),
            arrowprops=dict(arrowstyle="->"))

ax.set_xlabel("velocità $V$ [m/s]")              # simbolo E unita'
ax.set_ylabel("potenza [kW]")
ax.set_title("Curva di potenza del velivolo di riferimento")
ax.grid(alpha=0.3)                               # griglia leggera
ax.legend(loc="upper left")
plt.savefig("potenza.png", dpi=150, bbox_inches="tight")
```

Osservazione

Si noti `ax.annotate`: la posizione `xy` è il punto indicato dalla freccia, `xytext` è dove sta il testo. E si noti che l'etichetta dell'annotazione è *calcolata* dai dati (`Pn[i]:.1f`), non scritta a mano: se i dati cambiano, l'annotazione si aggiorna da sola. Un grafico con numeri scritti a mano è un grafico che prima o poi mentirà.

6.2 Famiglie di curve: il confronto al variare di un parametro

La domanda tecnica più frequente non è «com'è la curva?» ma «*come cambia* la curva al variare di un parametro?». Il rateo di salita al variare della quota, la polare al variare dell'allungamento, l'autonomia al variare del peso: in tutti questi casi la risposta è una *famiglia* di curve sullo stesso grafico, e il problema diventa distinguerle. Due strumenti: un ciclo che disegna e assegna la `label`, e una *mappa di colori* che codifica il parametro in una progressione cromatica ordinata.

famiglie.py (parte centrale)

```
from matplotlib import cm
from atmosfera20 import isa                      # riuso, come sempre

quote  = [0, 1500, 3000, 4500, 6000]
colori = cm.viridis(np.linspace(0.05, 0.85, len(quote)))

fig, ax = plt.subplots(figsize=(6.2, 3.8))
for h, c in zip(quote, colori):
    T, p, rho = isa(h)
    sigma = rho/1.225
    Vs = np.sqrt(2*W/(rho*S*1.6))
    V  = np.linspace(Vs*1.02, 70, 200)
    CL = 2*W/(rho*S*V**2)
    Pn = 0.5*rho*V**3*S*(CD0 + k*CL**2)
    RC = (0.75*90e3*sigma - Pn)/W                # motore aspirato
    ax.plot(V, RC, color=c, lw=1.7, label=f"h = {h} m")
ax.axhline(0, color="gray", lw=0.8)              # la linea dello zero
ax.legend()
```

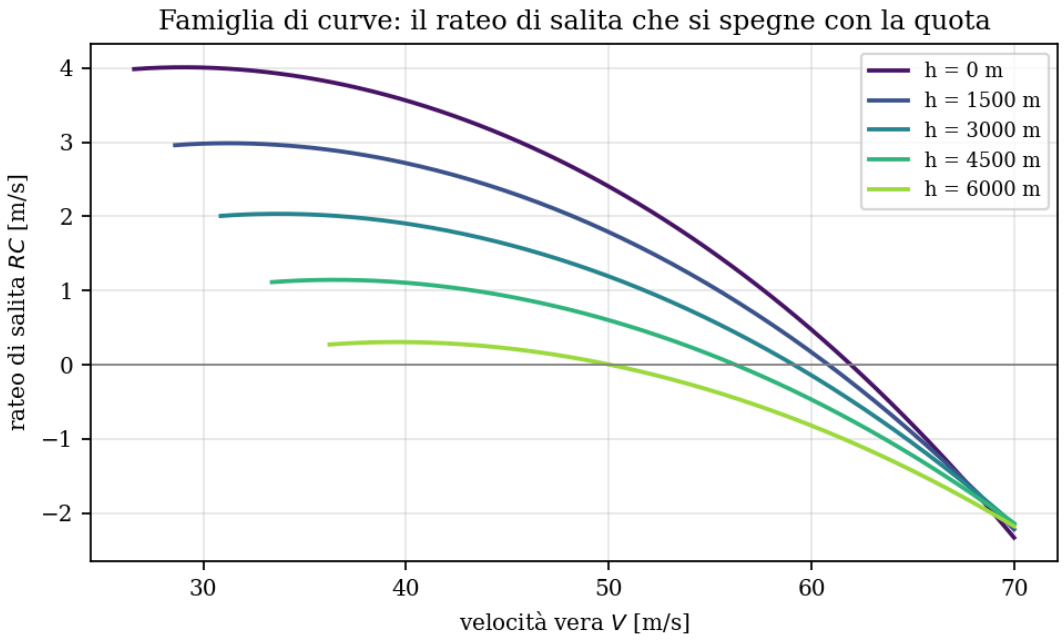


Figura 6.2. Cinque quote, una storia sola: il rateo di salita si spegne salendo, perché la potenza del motore aspirato cala con σ mentre quella necessaria cresce. La progressione dei colori (mappa *viridis*) rende leggibile l'ordine delle quote senza consultare la legenda curva per curva.

Lettura tecnica. Il grafico risponde a tre domande in un colpo solo: quanto vale RC_{max} a ogni quota (i massimi delle curve), a quale velocità conviene salire (le ascisse dei massimi, che crescono con la quota), e dove sta la *quota di tangenza* (dove l'intera curva scenderebbe sotto la linea dello zero — disegnarla con `axhline` non è un ornamento, è il riferimento fisico del problema). Tre risposte che dalla tabella dei numeri nessuno estrarrebbe a colpo d'occhio: ecco perché si disegna.

Osservazione

La mappa *viridis* non è una scelta estetica: è percettivamente uniforme (passi uguali del parametro appaiono come passi cromatici uguali), resta leggibile in scala di grigi e per i daltonici. Per famiglie ordinate da un parametro usate sempre una mappa sequenziale come questa; riservate i colori «a caso» alle curve di natura diversa fra loro.

6.3 Pannelli di sottografici

Quando le grandezze da mostrare hanno unità diverse, comprimerle in un solo riquadro produce pasticci; la soluzione pulita è il *pannello* di sottografici:

```
fig, assi = plt.subplots(2, 2, figsize=(7.6, 5.4))
```

restituisce una matrice 2×2 di assi, e `assi[0, 1]` si usa come qualunque `ax`. Le opzioni `sharex=True` / `sharey=True` agganciano le scale dei riquadri, così che l'occhio possa confrontare le colonne senza rifare la taratura a ogni passaggio.

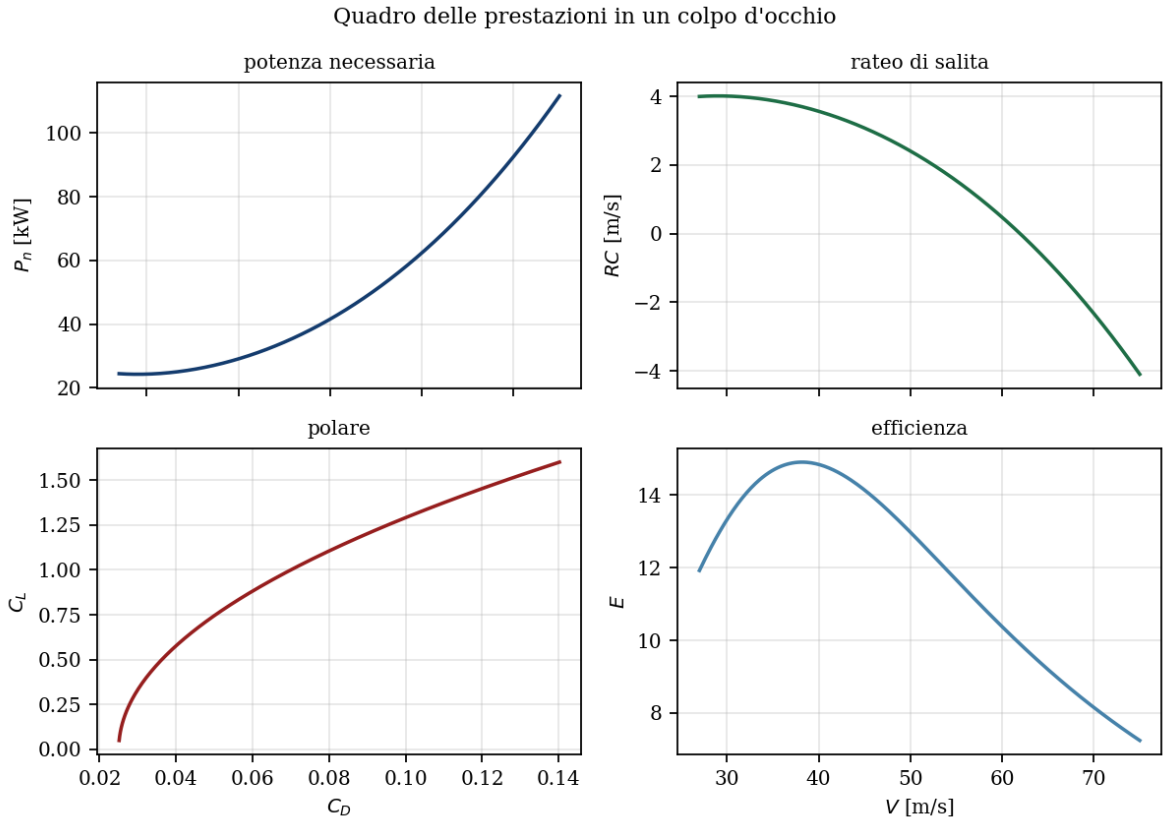


Figura 6.3. Il quadro delle prestazioni del velivolo di riferimento in quattro riquadri coordinati: potenza, rateo di salita, polare, efficienza. Un pannello ben composto è la pagina di sintesi di una relazione.

Osservazione

Regola di composizione: nei riquadri superiori si possono nascondere le etichette dell'asse x (`a.tick_params(labelbottom=False)`) quando coincidono con quelle dei riquadri sottostanti — meno inchiostro, più dati. È il principio guida di tutta la grafica tecnica: ogni segno sul foglio deve pagare l'informazione che occupa.

6.4 Due grandezze, due scale: il doppio asse

Capita di voler sovrapporre due grandezze legate fisicamente ma con unità e ordini di grandezza diversi: potenza necessaria e consumo orario, quota e temperatura, portanza e momento. Lo strumento è `ax2 = ax1.twinx()`: un secondo asse verticale che condivide l'asse x del primo.

`doppioasse.py (parte centrale)`

```
fig, ax1 = plt.subplots(figsize=(6.2, 3.7))
l1, = ax1.plot(V, Pn/1000, color="navy", lw=1.8, label="$P_n$ [kW]")
ax1.set_ylabel("potenza necessaria [kW]", color="navy")
ax1.tick_params(axis="y", labelcolor="navy")

ax2 = ax1.twinx()                          # secondo asse y, stessa x
l2, = ax2.plot(V, kg_h, color="firebrick", ls="--", lw=1.8,
               label="consumo [kg/h]")
ax2.set_ylabel("consumo orario [kg/h]", color="firebrick")
ax2.tick_params(axis="y", labelcolor="firebrick")

ax1.legend(handles=[l1, l2], loc="upper left")   # legenda unificata
```

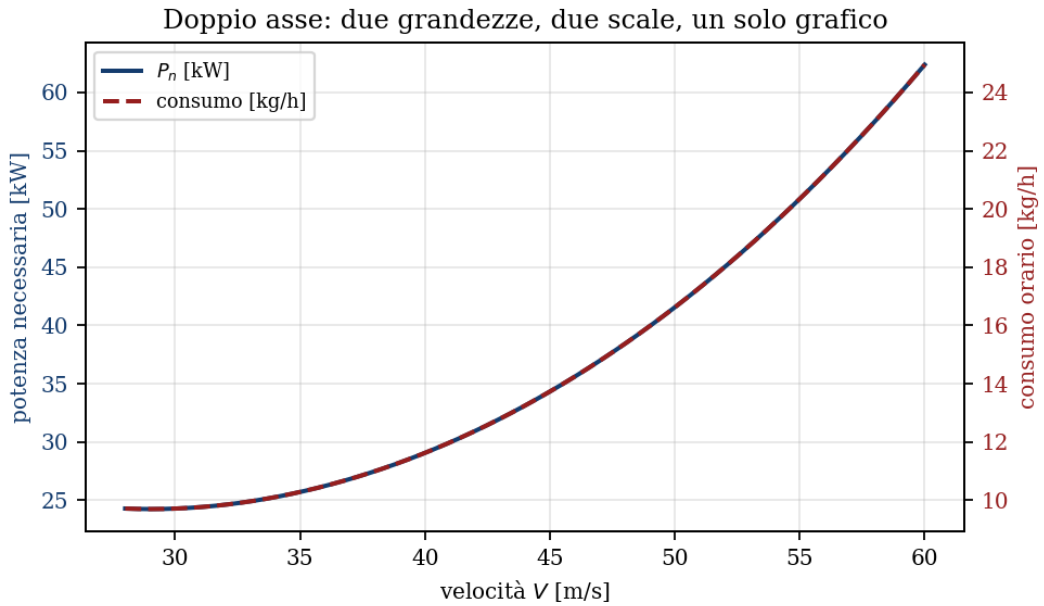


Figura 6.4. Potenza necessaria e consumo orario sullo stesso grafico: assi, etichette e curve condividono il colore, così l'occhio associa ogni curva alla propria scala senza ambiguità.

La convenzione che salva il lettore. Con due scale verticali l'ambiguità è in agguato: quale curva si legge su quale asse? La soluzione è cromatica e si vede nel codice: ogni asse porta il *colore della propria curva*

(etichetta e tacche comprese). E la legenda va unificata a mano passando le maniglie `l1, l2`, perché ogni asse ne genererebbe una propria. Ultima avvertenza: il doppio asse si usa per *due* grandezze, mai per tre — oltre, meglio un pannello.

6.5 Scale logaritmiche

Quando una grandezza varia di ordini di grandezza, la scala lineare schiaccia tutto il comportamento interessante in un angolo. La scala logaritmica (`ax.semilogy`, `ax.semilogx`, `ax.loglog`) restituisce leggibilità e — soprattutto — trasforma le leggi esponenziali in *rette*, rendendole riconoscibili a vista.

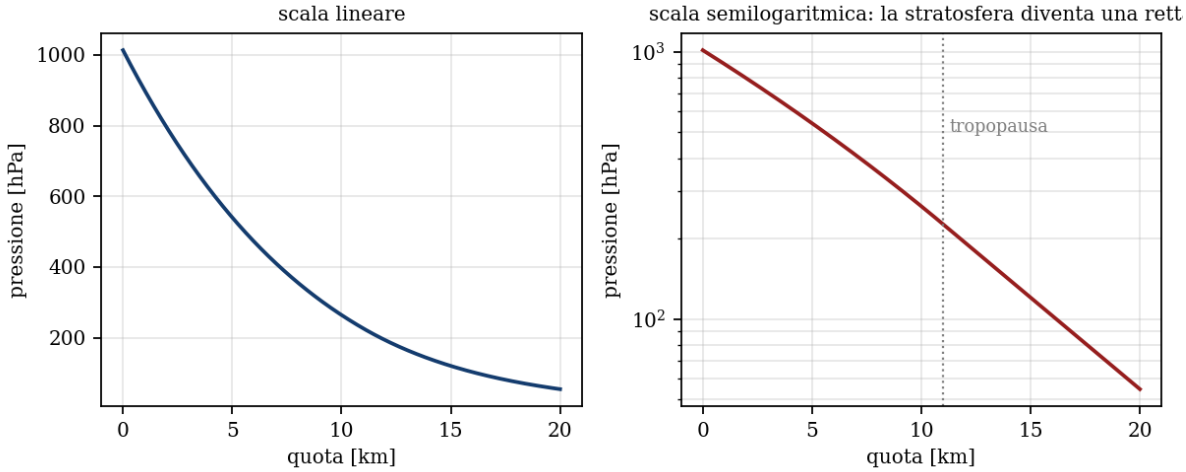


Figura 6.5. La pressione ISA da 0 a 20 km nelle due scale: in quella lineare la curva «muore» verso l'alto; in quella semilogaritmica la stratosfera isoterma — legge esponenziale, scheda 9.3 — diventa esattamente una retta oltre la tropopausa.

Lettura tecnica. Il pannello di destra è una *dimostrazione grafica*. Nella stratosfera isoterma vale

$$p = p_{11}\, e^{-g(h-11000)/(RT_s)} \qquad \Longrightarrow \qquad \ln p \text{ lineare in } h,$$

e la retta oltre gli 11 km lo certifica a vista. Nella troposfera la legge è una potenza, non un esponenziale, e infatti il tratto è lievemente curvo anche in scala semilog. Riconoscere una legge dalla forma che assume nella scala giusta (retta in semilog → esponenziale; retta in log–log → potenza) è un'abilità da laboratorio che vi servirà per tutta la carriera — dalle curve di fatica dei materiali, tracciate da sempre in log–log, agli spettri di turbolenza.

Prova tu

Tracciate in scala log–log il numero di Reynolds contro la quota (dati della scheda 9.8) e, accanto, la resistenza d'attrito laminare $C_f = 1{,}328/\sqrt{Re}$ contro Re da 10^5 a 10^7: nella scala giusta la seconda è una retta di pendenza $-1/2$. Misurate la pendenza dal grafico con due punti e verificate.

6.6 La figura da relazione: annotare fino in fondo

Mettiamo tutto insieme su un caso vero: il diagramma di manovra della scheda 10.9, portato al livello di una figura da allegare alla relazione d'esame. Gli ingredienti nuovi sono tre: `fill_between` per campire l'inviluppo ammesso, `axvline` per le velocità notevoli, e annotazioni con valori *calcolati*.

manovra_annotata.py (parte centrale)

```
n_pos = np.minimum(0.5*rho*V**2*S*CLmax/W, n_max)   # ramo di stallo
ax.fill_between(V, n_neg, n_pos, color="seagreen",
                alpha=0.10, label="inviluppo ammesso")
ax.plot(V, n_pos, lw=1.9); ax.plot(V, n_neg, lw=1.9)
ax.axvline(VA, color="firebrick", ls="--")
ax.annotate(f"$V_A$ = {VA:.1f} m/s", xy=(VA, n_max),
            xytext=(56, 3.1), arrowprops=dict(arrowstyle="->"))
ax.annotate("stallo: la struttura\ne' protetta",
            xy=(38, 2.05), xytext=(12, 2.6),
            arrowprops=dict(arrowstyle="->"), fontsize=8)
```

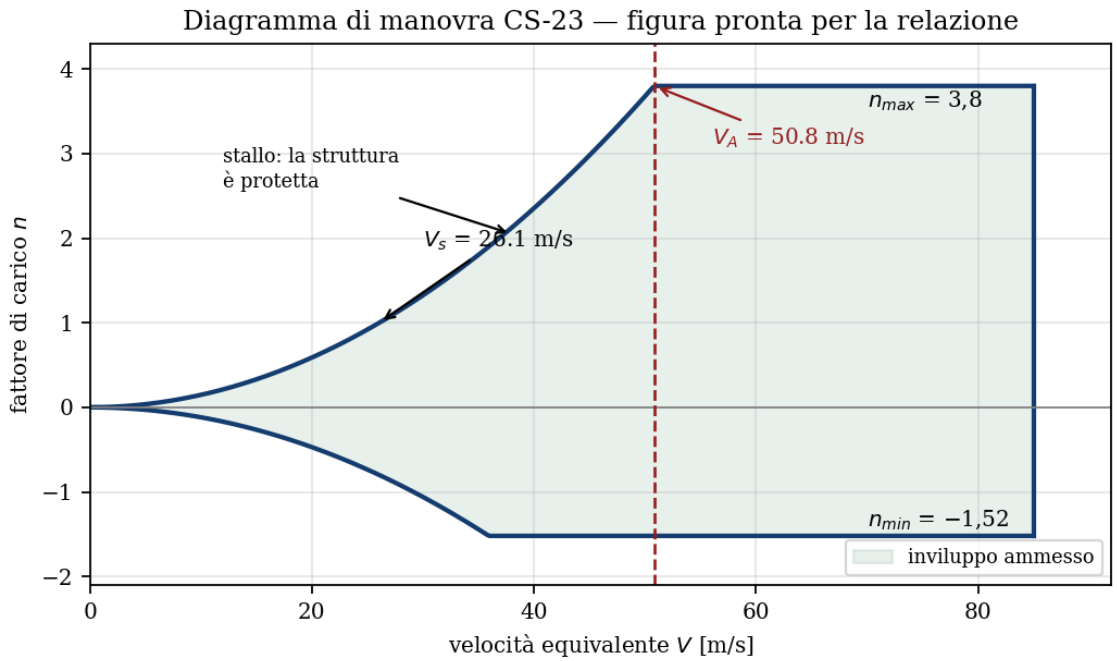


Figura 6.6. Il diagramma di manovra pronto per la relazione: area ammessa campita, ramo di stallo, V_A e V_s annotate con i valori calcolati, limiti n_{max} e n_{min} etichettati. Ogni elemento del disegno risponde a una domanda che l'esaminatore farebbe.

Prompt pronto all'uso

Genera con Matplotlib la figura del diagramma di manovra CS-23 per la relazione tecnica. Modello: ramo di stallo positivo $n = \frac{1}{2}\rho V^2 S C_{Lmax}/W$ troncato a n_{max} con np.minimum, ramo negativo analogo troncato a n_{min}, chiusura verticale a V_D. Dati: il velivolo di riferimento, $n_{max} = 3{,}8$, $n_{min} = -1{,}52$, $V_D = 85$ m/s. Vincoli di formato: area ammessa campita con fill_between e alpha 0,10; V_A evidenziata con axvline tratteggiata e annotata con il valore CALCOLATO nel testo dell'annotazione (f-string, non numero scritto a mano); etichette degli assi con simbolo e unità; griglia con alpha 0,3; salvataggio a 150 dpi con bbox_inches="tight". Controllo visivo: la parabola di stallo deve passare per $(V_s, 1)$ e toccare n_{max} esattamente in V_A.

6.7 Qualità di stampa e stile

Obiettivo	Strumento
Stile uniforme per tutto il documento	`plt.rcParams.update({...})`
Carattere coerente con la relazione	`"font.family": "serif"`
Dimensione pensata per la pagina	`figsize=(6.4, 4)` ($\simeq$ testo A4)
Nitidezza in stampa	`dpi=150` (300 per la stampa vera)
Margini senza sprechi	`bbox_inches="tight"`
Distinguere le curve anche in B/N	`ls="-"`, `ls=":"`, marcatori
Formato vettoriale (scala infinita)	`plt.savefig("fig.pdf")`

Osservazione

Due insidie ricorrenti. Prima: una relazione può essere stampata in bianco e nero — se le curve si distinguono *solo* per colore, in stampa diventano indistinguibili; differenziate anche il tratto (`ls`) o i marcatori. Seconda: la scelta `figsize` determina la dimensione *relativa* dei caratteri — una figura generata enorme e poi rimpicciolita in pagina ha etichette illeggibili; generatela già della larghezza a cui verrà stampata. E ricordate la regola geometrica del capitolo 8: `set_aspect("equal")` è obbligatorio per profili, vettori e piante alari, controproducente per i diagrammi funzionali.

6.8 Oltre Matplotlib: uno sguardo alle librerie affini

Matplotlib è la fondazione, e per la grafica tecnica di questo corso basta e avanza. Ma conviene sapere che cosa c'è intorno, perché lo incontrerete presto. **pandas**, la libreria dei dati tabellari, incorpora scorciatoie di tracciamento (`dataframe.plot()`) che producono figure Matplotlib direttamente dalle colonne di una tabella: naturale quando i dati arrivano da un file CSV di laboratorio. **seaborn** è costruita sopra Matplotlib e automatizza la grafica *statistica* — distribuzioni, regressioni, matrici di correlazione: utile quando dalle prove in galleria tornano centinaia di punti dispersi anziché curve pulite. **Plotly** genera grafici *interattivi* per il browser (zoom, ispezione dei valori col puntatore, rotazione delle superfici 3D): è lo strumento giusto quando il grafico va esplorato, o pubblicato in una pagina web come quelle che il vostro istituto già usa. La buona notizia: la grammatica appresa qui — assi, etichette, famiglie, scale — si trasferisce identica; cambia la sintassi, non il mestiere.

Dialoga con l'IA

Un dialogo di collaudo per chiudere: consegnate all'assistente il vostro `famiglie.py` e chiedete *"Convertilo in un grafico interattivo Plotly con le stesse curve, le stesse etichette e le unità sugli assi"*. Poi collaudate come fareste con un calcolo: passando il puntatore sulla curva di quota zero, il massimo deve leggersi 4,0 m/s a 28 m/s — gli stessi valori della scheda 10.3. Un grafico, per quanto interattivo e brillante, si valida come un numero: contro un valore noto.

6.9 Lista di controllo del grafico tecnico

	Verifica
☐	Etichette degli assi con simbolo *e* unità di misura
☐	Titolo che dichiara che cosa mostra la figura
☐	Legenda presente se le curve sono più di una
☐	Griglia leggera (`alpha` $\simeq 0{,}3$), mai invadente
☐	Curve distinguibili anche in bianco e nero
☐	Famiglie ordinate con mappa di colori sequenziale
☐	Annotazioni con valori calcolati, non trascritti a mano
☐	Scala giusta per la fisica (log per gli esponenziali)
☐	`set_aspect("equal")` se il disegno è geometrico
☐	Riferimenti fisici disegnati (lo zero, i limiti, le ammissibili)
☐	Salvataggio con `dpi` adeguato e `bbox_inches="tight"`
☐	La figura è rigenerata dall'ultima versione dei dati

Capitolo 7

L'IA come assistente di calcolo: il metodo

In questo capitolo imparerai a

- scrivere un prompt tecnico completo (dati, unità, modello, risultato atteso);
- applicare il ciclo *genera – leggi – valida – correggi*;
- riconoscere gli errori tipici del codice generato dall'IA.

7.1 Perché un capitolo di metodo

In ufficio tecnico nessun risultato entra in una relazione senza verifica, chiunque — o qualunque cosa — l'abbia prodotto. L'IA è un assistente rapidissimo e instancabile, ma *statistico*: produce il codice più plausibile, non necessariamente quello corretto. Chi firma il calcolo siete voi. Da qui il metodo del corso, in quattro passi.

Il ciclo GLVC

Genera: chiedi all'IA il programma con un prompt completo. **Leggi**: percorri il codice riga per riga finché sai spiegare ogni istruzione. **Valida**: confronta almeno un output con un calcolo manuale o un valore tabellato (ISA, manuale, prova ministeriale). **Correggi**: se qualcosa non torna, individua tu la riga sospetta e chiedi una correzione *mirata*, non una riscrittura.

7.2 Anatomia di un buon prompt tecnico

Confrontate i due prompt seguenti per lo stesso problema.

Prompt debole

"Fammi un programma sull'atmosfera."

Prompt tecnico

"Scrivi un programma Python per studenti che, data una quota h in metri (da input, valida solo se $0 \le h \le 11000$), calcoli temperatura, pressione e densità con il modello ISA della troposfera: $T = T_0 - \lambda h$, $p = p_0(T/T_0)^{g/(R\lambda)}$, $\rho = p/(RT)$, con $T_0 = 288{,}15$ K, $p_0 = 101\,325$ Pa, $\lambda = 0{,}0065$ K/m, $R = 287{,}05$ J/(kg K). Stampa i risultati con unità SI e 4 decimali sulla densità. Usa solo variabili con nomi fisici e commenta le unità. *Controllo*: a $h = 5000$ m deve risultare $\rho \approx 0{,}7361$ kg/m^3."

Il secondo prompt contiene le cinque componenti che pretenderemo sempre: **(1)** il modello fisico esplicito, **(2)** i dati con le unità, **(3)** i vincoli di validità, **(4)** il formato dell'output, **(5)** *un valore di controllo per il collaudo*. Il punto (5) è la vostra assicurazione: costringe l'IA — e voi — a un riscontro oggettivo immediato.

7.3 Gli errori tipici del codice generato

L'esperienza d'aula ne indica quattro ricorrenti; imparate a cercarli per primi.

1. **Unità incoerenti**: quote in piedi dove il modello vuole metri, velocità in nodi dentro $q = \frac{1}{2}\rho V^2$. Sintomo: risultati sbagliati di fattori 3,28 o 1,94.
2. **Costanti "a memoria"**: $R = 287$ invece di 287,05, $g = 9{,}81$ invece di 9,80665. Innocuo quasi sempre, ma dovete *saperlo*, perché sposta le ultime cifre rispetto alle tabelle.
3. **Domini di validità ignorati**: la formula della troposfera applicata a 15 000 m, la polare parabolica usata oltre lo stallo. Il codice gira, il numero è privo di significato: l'errore più pericoloso, perché silenzioso.
4. **Eccesso di sofisticazione**: classi, gestione d'eccezioni e librerie esotiche per un problema da dieci righe. Chiedete esplicitamente: "solo costrutti di base, livello principiante".

Attenzione

La regola di consegna del corso: *ogni programma consegnato deve riportare, in un commento finale, la validazione svolta* — quale grandezza è stata controllata, contro quale riferimento, con quale scarto. Un programma senza validazione è un calcolo senza collaudo: formalmente esiste, tecnicamente non vale nulla.

7.4 L'interrogazione alla rovescia

L'IA serve anche per *studiare*, non solo per produrre. Tre usi consigliati: farsi spiegare un codice altrui riga per riga ("commenta ogni riga, poi fammi tre domande per verificare che io abbia capito"); farsi generare varianti d'esercizio ("stesso problema, dati diversi, con soluzione separata"); farsi interrogare ("fammi cinque domande sul modello ISA, una alla volta, correggendo le mie risposte"). In tutti e tre i casi l'IA lavora *per* la vostra testa, non al posto suo.

Prova tu

1. Riscrivete come prompt tecnico completo (cinque componenti) questa richiesta debole: "fammi un programma sulla portanza".
2. Fatevi generare il programma dal vostro prompt e compilate la scheda di collaudo: valore controllato, riferimento, scarto.
3. Scambiatevi i programmi generati con un compagno e trovate almeno un difetto ciascuno (unità, costanti, dominio, leggibilità).

Capitolo 8

Vettori con Python: calcolare e vedere

In questo capitolo imparerai a

- rappresentare un vettore come `array` NumPy e calcolarne modulo, versore e angolo;
- eseguire somma, differenza, prodotto scalare e prodotto vettoriale per componenti;
- *disegnare* vettori e operazioni con Matplotlib, per vedere ciò che si è calcolato;
- calcolare momenti, verificare il teorema di Varignon e ridurre un sistema di forze a un polo;
- costruire una piccola libreria personale, `vettori2d.py`, e collaudarla sugli esercizi del testo di teoria.

Nota

Questo capitolo è il ponte fra la monografia *Teoria dei vettori* — che d'ora in poi chiameremo semplicemente *il testo di teoria* — e le parti annuali che seguono. Non introduce alcuna nuova fisica: prende la teoria già studiata e le fornisce uno strumento di calcolo e, soprattutto, uno strumento di *visione*. Ogni programma di questo capitolo è collaudato su un esercizio svolto del testo di teoria, di cui riproduce il risultato cifra per cifra: chi ha il testo accanto può verificare ogni riga. È il modo più onesto di introdurre uno strumento nuovo — farlo rispondere a domande di cui conosciamo già la risposta.

8.1 Dal simbolo all'array

Il testo di teoria scrive un vettore del piano come $\mathbf{A} = (A_x, A_y)$, e uno dello spazio come $\mathbf{A} = (A_x, A_y, A_z)$. Python, con NumPy, adotta esattamente la stessa scrittura:

la traduzione elementare

```
import numpy as np

A = np.array([3.0, 4.0])          # vettore del piano
B = np.array([1.0, 2.0, -2.0])    # vettore dello spazio
```

Le parentesi quadre delimitano la lista delle componenti; `np.array` la trasforma in un oggetto su cui le operazioni agiscono *componente per componente*, come richiede l'algebra vettoriale.

Osservazione

Scrivete sempre `3.0` e non `3`. Un array di interi propaga gli interi: `np.array([3,4])/2` restituisce (1,5; 2,0) e va bene, ma altre operazioni possono troncare silenziosamente. Il punto decimale costa un carattere e previene una classe intera di errori muti — quelli che non danno messaggio, e sono i peggiori.

8.2 Le tre domande fondamentali

Dato un vettore, il testo di teoria pone sempre tre domande: quanto vale (*modulo*), dove punta (*versore*), che angolo forma (*direzione*). Tre domande, tre funzioni.

Formula chiave

$$\|\mathbf{A}\| = \sqrt{A_x^2 + A_y^2}, \qquad \hat{a} = \frac{\mathbf{A}}{\|\mathbf{A}\|}, \qquad \alpha = \operatorname{atan2}(A_y, A_x)$$

Il modulo è non negativo per definizione (è una lunghezza); il versore ha modulo unitario — e questa è la sua verifica gratuita.

Prompt pronto all'uso

Scrivi un programma Python che, dato un vettore del piano, ne calcoli modulo, versore e angolo con l'asse x. Modello: modulo con np.linalg.norm, versore per divisione, angolo con math.atan2 (NON math.atan: spiega la differenza in un commento). Dati: $\mathbf{A} = (3, 4)$. *Formato: modulo e angolo con due decimali, angolo in gradi. Controllo: il modulo deve valere esattamente 5 (terna pitagorica 3–4–5), l'angolo* 53,13 *gradi e il versore deve avere modulo 1 — stampa anche quest'ultima verifica.*

vettore_base.py

```python
# vettore_base.py -- modulo, versore, angolo di un vettore
import numpy as np, math

A = np.array([3.0, 4.0])              # il vettore A = (3, 4)

modulo = np.linalg.norm(A)            # ||A||: sempre non negativo
versore = A/modulo                    # a-cappuccio: modulo unitario
alfa = math.degrees(math.atan2(A[1], A[0]))   # atan2, MAI atan

print(f"A       = {A}")
print(f"modulo  = {modulo:.3f}")
print(f"versore = {versore}   (controllo: {np.linalg.norm(versore):.6f})")
print(f"angolo  = {alfa:.2f} gradi")
```

Output

```
A       = [3. 4.]
modulo  = 5.000
versore = [0.6 0.8]   (controllo: 1.000000)
angolo  = 53.13 gradi
```

Validazione. È l'esercizio E1 del testo di teoria nella sua forma più semplice: $\sqrt{3^2 + 4^2} = 5$ esatto e $\hat{a} = (0{,}6;\ 0{,}8)$, il cui modulo di controllo vale 1,000000 ✓. Terne pitagoriche come 3–4–5, 6–8–10 e 2–3–6 (nello spazio) sono il vostro laboratorio: danno moduli interi e permettono la verifica a mente.

Osservazione

La funzione `math.degrees` è indispensabile: Python, come ogni libreria scientifica, lavora *in radianti*. Dimenticarla produce un 0,927 al posto di 53,13: un numero plausibile e completamente sbagliato. Convertite sempre all'ultimo istante, in stampa, e tenete i radianti nei calcoli.

8.3 Algebra per componenti

Il paragrafo 5.3 del testo di teoria stabilisce che sommare vettori significa sommare le componenti omologhe. In NumPy questa frase diventa un carattere: il segno +.

somma_componenti.py

```python
# somma_componenti.py -- algebra dei vettori per componenti
import numpy as np, math

A = np.array([3.0, 4.0])
B = np.array([-1.0, 2.0])

for nome, v in [("A", A), ("B", B), ("A+B", A+B), ("A-B", A-B)]:
    mod = np.linalg.norm(v)
    ang = math.degrees(math.atan2(v[1], v[0]))
    print(f"{nome:4s} = {str(v):14s} modulo = {mod:6.3f}   angolo =
    → {ang:7.2f} gradi")

print(f"\n||A|| + ||B|| = {np.linalg.norm(A)+np.linalg.norm(B):.3f}"
      f"   ma  ||A+B|| = {np.linalg.norm(A+B):.3f}")
```

Output

```
A    = [3. 4.]         modulo =  5.000   angolo =   53.13 gradi
B    = [-1.  2.]       modulo =  2.236   angolo =  116.57 gradi
A+B  = [2. 6.]         modulo =  6.325   angolo =   71.57 gradi
A-B  = [4. 2.]         modulo =  4.472   angolo =   26.57 gradi

||A|| + ||B|| = 7.236   ma  ||A+B|| = 6.325
```

Validazione. È l'esercizio svolto 5.1 del testo di teoria, riprodotto interamente: $\mathbf{A}+\mathbf{B}=(2,6)$, $\mathbf{A}-\mathbf{B}=(4,2)$, moduli 5, 2,236, 6,325 e angoli 53,13, 116,57, 71,57 gradi ✓. L'ultima riga stampa l'avvertenza con cui il testo chiude l'esercizio: i moduli *non* si sommano. Qui il programma non calcola soltanto: dimostra.

Osservazione

L'angolo di $\mathbf{B}$ è 116,57 gradi, nel secondo quadrante. Con `math.atan(2/-1)` si otterrebbe $-63{,}43$ gradi — la direzione opposta. È l'errore che il testo di teoria segnala nel riquadro rosso del paragrafo 5.2, e che nei problemi di equilibrio significa invertire il verso di una reazione vincolare.

8.4 Vedere i vettori: il comando `quiver`

Fin qui abbiamo calcolato. Ora facciamo la cosa che rende questo capitolo diverso da un esercizio di aritmetica: *disegniamo*. Matplotlib traccia frecce con `quiver`, e il disegno è il collaudo più rapido che esista — un vettore sbagliato *si vede*.

Formula chiave

La sintassi da imparare a memoria è

```
ax.quiver(x0, y0, dx, dy, angles="xy", scale_units="xy", scale=1)
```

dove (x_0, y_0) è la coda della freccia e (dx, dy) sono le componenti. Le tre opzioni finali sono obbligatorie:

senza di esse Matplotlib riscala autonomamente le frecce e il disegno mente.

Prompt pronto all'uso

Scrivi un programma Python che disegni due vettori del piano e la loro somma con la regola del parallelogramma. Modello: matplotlib con ax.quiver e le opzioni angles="xy", scale_units="xy", scale=1 per evitare il riscalamento automatico; lati del parallelogramma tratteggiati con ax.plot. Dati: $\mathbf{A} = (3, 4)$, $\mathbf{B} = (-1, 2)$. *Vincoli di formato: ax.set_aspect("equal") obbligatorio (altrimenti gli angoli risultano deformati), griglia leggera, etichette a metà di ciascuna freccia, figura salvata in PNG a 150 dpi. Controllo visivo: la freccia verde deve chiudere il parallelogramma e terminare nel punto (2,6).*

disegna.py

```python
# disegna.py -- visualizzare vettori con quiver
import numpy as np
import matplotlib.pyplot as plt

A = np.array([3.0, 4.0])
B = np.array([-1.0, 2.0])
S = A + B

fig, ax = plt.subplots(figsize=(5, 4))

# quiver(x_coda, y_coda, dx, dy): le tre opzioni angles/scale_units/scale
# servono a NON far riscalare le frecce da Matplotlib.
for v, colore, nome in [(A, "navy", "A"), (B, "firebrick", "B"),
                        (S, "seagreen", "A+B")]:
    ax.quiver(0, 0, v[0], v[1], angles="xy", scale_units="xy",
              scale=1, color=colore, width=0.008)
    ax.text(v[0]*0.55, v[1]*0.55 + 0.15, nome, color=colore, fontsize=12)

# lati tratteggiati del parallelogramma
ax.plot([A[0], S[0]], [A[1], S[1]], "--", color="gray", lw=1)
ax.plot([B[0], S[0]], [B[1], S[1]], "--", color="gray", lw=1)

ax.set_xlim(-2, 4); ax.set_ylim(-1, 7)
ax.set_aspect("equal")          # INDISPENSABILE: angoli non deformati
ax.grid(alpha=0.3)
ax.set_xlabel("x"); ax.set_ylabel("y")
ax.set_title("Regola del parallelogramma")
plt.savefig("parallelogramma.png", dpi=150, bbox_inches="tight")
print("figura salvata: parallelogramma.png")
```

Osservazione

`ax.set_aspect("equal")` non è un dettaglio estetico: senza di esso gli assi hanno scale diverse, un angolo di 45 gradi appare di 70, e il disegno — che dovrebbe essere il vostro strumento di controllo — diventa esso stesso una fonte di errore. Un profilo alare disegnato senza `equal` sembra sempre più spesso di quanto è: lo avete già incontrato nel capitolo 5.

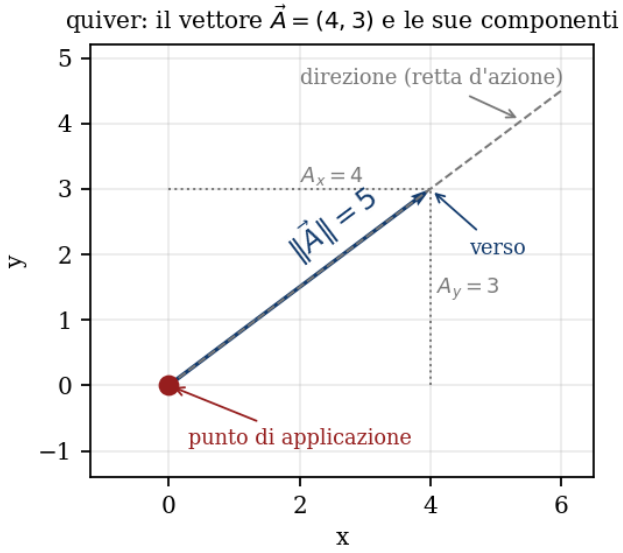


Figura 8.1. Anatomia di un vettore disegnata con `quiver`: punto di applicazione, retta d'azione, verso, modulo e componenti. Confrontatela con la Figura 2 del testo di teoria.

8.5 Parallelogramma, poligono e teorema di Carnot

Il testo di teoria presenta due costruzioni grafiche equivalenti per la somma: il parallelogramma (paragrafo 4.2) e il poligono testa–coda (paragrafo 4.3). La differenza, in Python, è solo il punto in cui si mette la coda della freccia.

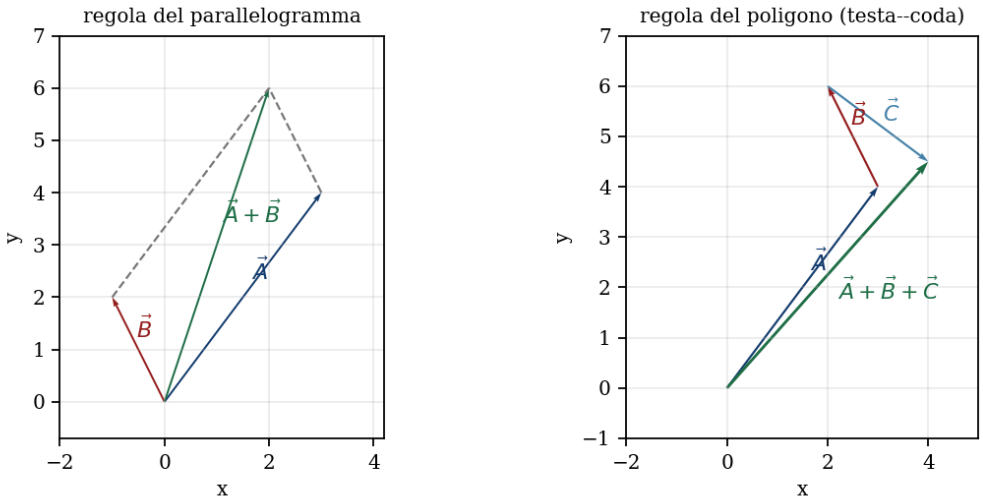


Figura 8.2. Le due costruzioni grafiche generate dallo stesso programma: a sinistra il parallelogramma (tutte le code nell'origine), a destra il poligono (ogni coda sulla punta precedente). La risultante è identica.

Vediamo ora il primo esercizio svolto del testo di teoria (il 4.1), in cui due cavi tirano il gancio di traino di un aliante. Il controllo si esegue con il teorema di Carnot, cioè per una via del tutto indipendente dalle componenti.

Formula chiave

Per due vettori di moduli F_1, F_2 che formano fra loro l'angolo γ, il modulo della risultante vale

$$R = \sqrt{F_1^2 + F_2^2 + 2F_1F_2\cos\gamma}$$

Il segno + davanti al doppio prodotto (anziché il $-$ del teorema del coseno per i triangoli) discende dal fatto che l'angolo interno del triangolo delle forze è il supplementare di γ.

Prompt pronto all'uso

Scrivi un programma Python che calcoli la risultante di due forze date in forma polare. Modello: conversione da (modulo, angolo) a componenti con una funzione dedicata, somma per componenti, poi modulo e angolo della risultante. Dati: $F_1 = 300$ N a 0 gradi, $F_2 = 200$ N a 60 gradi. Formato: componenti, modulo con un decimale, angolo con un decimale. Controllo indipendente: ricalcola il modulo con il teorema di Carnot $R = \sqrt{F_1^2 + F_2^2 + 2F_1F_2\cos\gamma}$ e stampa entrambi — devono coincidere a 435,9 N.

carnot.py

```python
# carnot.py -- risultante di due forze: componenti vs teorema di Carnot
import numpy as np, math

F1_mod, F1_ang = 300.0,  0.0     # [N], [gradi sull'orizzontale]
F2_mod, F2_ang = 200.0, 60.0

def polare(modulo, gradi):
    """Da (modulo, angolo in gradi) alle componenti cartesiane."""
    a = math.radians(gradi)
    return modulo*np.array([math.cos(a), math.sin(a)])

F1, F2 = polare(F1_mod, F1_ang), polare(F2_mod, F2_ang)
R = F1 + F2

print(f"F1 = ({F1[0]:7.2f}, {F1[1]:7.2f}) N")
print(f"F2 = ({F2[0]:7.2f}, {F2[1]:7.2f}) N")
print(f"R  = ({R[0]:7.2f}, {R[1]:7.2f}) N")
print(f"|R| = {np.linalg.norm(R):.1f} N   theta = "
      f"{math.degrees(math.atan2(R[1], R[0])):.1f} gradi")

# controllo indipendente: teorema di Carnot (del coseno)
gamma = math.radians(F2_ang - F1_ang)
R_carnot = math.sqrt(F1_mod**2 + F2_mod**2 + 2*F1_mod*F2_mod*math.cos(gamma))
print(f"controllo con Carnot: |R| = {R_carnot:.1f} N")
```

Output

```
F1 = ( 300.00,    0.00) N
F2 = ( 100.00,  173.21) N
R  = ( 400.00,  173.21) N
|R| = 435.9 N   theta = 23.4 gradi
controllo con Carnot: |R| = 435.9 N
```

Validazione. Esercizio svolto 4.1 del testo di teoria: $R \simeq 435{,}9$ N inclinata di 23,4 gradi ✓. Le due strade — componenti e Carnot — non condividono alcun passaggio di calcolo, eppure convergono sulla quarta cifra:

quando due derivazioni indipendenti coincidono, il risultato è dimostrato. Osservate inoltre che F_2 a 60 gradi ha componenti (100; 173,21): esattamente $200\cos 60^\circ = 100$ e $200\sin 60^\circ = 100\sqrt{3}$.

Prova tu

Generalizzate `polare()` a un elenco di forze qualsiasi e riscrivete l'esercizio 9.1 del testo di teoria (tre tiranti da 500 N a 0, 120 e 240 gradi). Dovete ottenere $\mathbf{R} = \mathbf{0}$: il nodo è in equilibrio. Attenzione a come si stampa quello zero — ne riparliamo nel paragrafo 8.14.

8.6 Il prodotto scalare: angoli, proiezioni, lavoro

Il prodotto scalare, dice il testo di teoria, «misura quanto due vettori vanno d'accordo». In NumPy è `np.dot`, e la sua doppia faccia — componenti e geometria — diventa una doppia via di calcolo, cioè un collaudo.

Formula chiave

$$\mathbf{A}\cdot\mathbf{B} = A_xB_x + A_yB_y = \|\mathbf{A}\|\,\|\mathbf{B}\|\cos\gamma$$

Da cui le due applicazioni operative: l'angolo fra due direzioni, $\cos\gamma = \mathbf{A}\cdot\mathbf{B}/(\|\mathbf{A}\|\|\mathbf{B}\|)$, e la proiezione di $\mathbf{A}$ su una direzione $\hat{u}$, $A_u = \mathbf{A}\cdot\hat{u}$.

Prompt pronto all'uso

Scrivi un programma Python sul prodotto scalare. Modello: np.dot per la forma per componenti, poi inversione della forma geometrica per ricavare l'angolo (usa math.acos); infine la proiezione di un vettore su un versore e il lavoro di una forza costante $L = \mathbf{F}\cdot\mathbf{s}$. Dati: $\mathbf{A} = (3, 4)$ e $\mathbf{B} = (4, 3)$ per l'angolo; $\mathbf{F} = (300, 200)$ N e $\mathbf{s} = (5, 0)$ m per il lavoro. Formato: angolo in gradi con due decimali, lavoro in joule. Controlli: l'angolo deve valere 16,26 gradi e il lavoro 1500 J, ricalcolato anche nella forma $\|F\|\|s\|\cos\gamma$.

`scalare.py`

```
# scalare.py -- prodotto scalare: angolo, proiezione, lavoro
import numpy as np, math

A = np.array([3.0, 4.0])
B = np.array([4.0, 3.0])

d = np.dot(A, B)                                  # forma per componenti
cos_g = d/(np.linalg.norm(A)*np.linalg.norm(B))   # forma geometrica invertita
print(f"A . B   = {d:.0f}")
print(f"cos g   = {cos_g:.4f}   ->  gamma =
    → {math.degrees(math.acos(cos_g)):.2f} gradi")

# proiezione di A sulla direzione di B
u = B/np.linalg.norm(B)
print(f"proiezione di A su B: {np.dot(A, u):.3f}")

# lavoro di una forza costante
F = np.array([300.0, 200.0])      # [N]
s = np.array([5.0, 0.0])          # [m]
L = np.dot(F, s)
print(f"\nlavoro L = {L:.0f} J")
g2 = math.degrees(math.atan2(F[1], F[0]))
print(f"controllo: |F| |s| cos g = "
```

```
    f"{np.linalg.norm(F)*np.linalg.norm(s)*math.cos(math.radians(g2)):.0f} J")
```

Output

```
A . B   = 24
cos g   = 0.9600   ->  gamma = 16.26 gradi
proiezione di A su B: 4.800

lavoro L = 1500 J
controllo: |F| |s| cos g = 1500 J
```

Validazione. Esercizi svolti 7.1 e 7.2 del testo di teoria: $\mathbf{A}\cdot\mathbf{B} = 24$, $\gamma \simeq 16{,}26$ gradi, e $L = 1500\,\text{J}$ per entrambe le vie ✓. La proiezione vale 4,8: dei 5 di modulo di **A**, la parte «utile» lungo **B** è 4,8, e la parte perpendicolare, che non compie lavoro, completa a 5 per Pitagora ($4{,}8^2 + 1{,}4^2 = 25$).

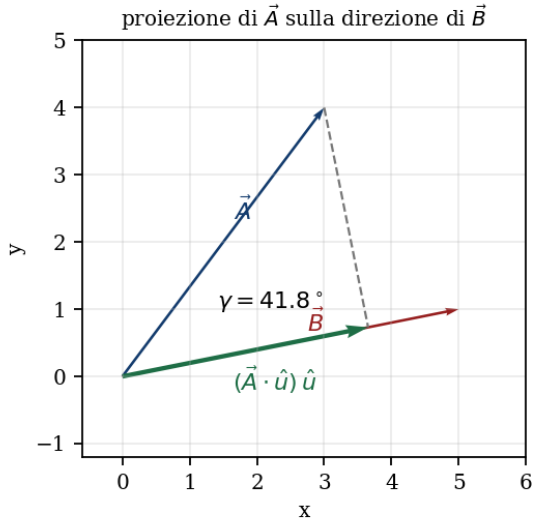


Figura 8.3. La proiezione di **A** sulla direzione di **B**: il segmento verde è $(\mathbf{A}\cdot\hat{u})\,\hat{u}$, e il tratteggio grigio è la componente perpendicolare, che nel lavoro non contribuisce.

Osservazione

L'ortogonalità si collauda con una sola riga: `abs(np.dot(A, B)) < 1e-9`. Non scrivete `np.dot(A, B) == 0`: in virgola mobile due vettori perpendicolari possono dare $2{,}2\times10^{-16}$ anziché zero esatto, e il confronto fallirebbe. Il confronto a tolleranza è la regola d'oro di ogni verifica numerica.

Prova tu

Riprendete l'esercizio E8 del testo di teoria, quello «a trabocchetto» sul lavoro della portanza. Scrivete un programma che, dato l'angolo di rampa γ, costruisca **L** perpendicolare alla velocità e calcoli $\mathbf{L}\cdot\mathbf{V}$ per γ da 0 a 15 gradi: deve risultare nullo *sempre*, in salita come in orizzontale. Poi ripetete con il peso: lì il prodotto scalare non è nullo, ed è negativo (lavoro resistente).

8.7 Il prodotto vettoriale: dal piano allo spazio

Il prodotto vettoriale è l'operazione da cui nascono i momenti. Il testo di teoria distingue nettamente il caso piano (dove il risultato ha la sola componente z) dal caso spaziale (dove servono tutte e tre): manteniamo la distinzione anche nel codice, perché in Python `np.cross` si comporta in modo diverso nei due casi.

Formula chiave

Nel piano, dove il risultato è diretto lungo z:

$$(\mathbf{A} \times \mathbf{B})_z = A_x B_y - A_y B_x$$

Il segno è un'informazione fisica: positivo se la rotazione da $\mathbf{A}$ verso $\mathbf{B}$ è antioraria, negativo se oraria. Nello spazio, il determinante simbolico del paragrafo 8.4 del testo di teoria.

Prompt pronto all'uso

Scrivi un programma Python sul prodotto vettoriale. Modello: nel piano definisci una funzione cross_z(A,B) che restituisce $A_xB_y - A_yB_x$ e ne interpreta il segno (antiorario/orario); nello spazio usa np.cross. Dati: $(3, 4)$ *e* $(4, 3)$ *per il piano;* $\mathbf{A} = (1, 2, 3)$ *e* $\mathbf{B} = (4, 5, 6)$ *per lo spazio. Formato: componenti e modulo. Controlli obbligatori: mostra che* $\mathbf{B} \times \mathbf{A}$ *ha segno opposto (non commutativo) e verifica con np.dot che il risultato spaziale sia ortogonale a entrambi i fattori — entrambi i prodotti scalari devono dare zero.*

`vettoriale.py`

```python
# vettoriale.py -- prodotto vettoriale nel piano e nello spazio
import numpy as np

def cross_z(A, B):
    """Componente z del prodotto vettoriale di due vettori del piano."""
    return A[0]*B[1] - A[1]*B[0]

# --- caso piano ---
A2 = np.array([3.0, 4.0])
B2 = np.array([4.0, 3.0])
print(f"(A x B)_z = {cross_z(A2, B2):.0f}"
      f"   -> rotazione {'antioraria' if cross_z(A2,B2) > 0 else 'oraria'}")
print(f"(B x A)_z = {cross_z(B2, A2):.0f}   -> segno opposto: NON
    → commutativo")

# --- caso spaziale ---
A = np.array([1.0, 2.0, 3.0])
B = np.array([4.0, 5.0, 6.0])
C = np.cross(A, B)
print(f"\nA x B = {C}   modulo = {np.linalg.norm(C):.2f}")

# collaudo: il risultato deve essere ortogonale a entrambi i fattori
print(f"controllo (AxB).A = {np.dot(C, A):.0f}    (AxB).B = {np.dot(C,
    → B):.0f}")
print(f"area del parallelogramma = {np.linalg.norm(C):.2f}")
```

```
Output
```

```
(A x B)_z = -7   -> rotazione oraria
(B x A)_z = 7   -> segno opposto: NON commutativo

A x B = [-3.  6. -3.]   modulo = 7.35
controllo (AxB).A = 0    (AxB).B = 0
area del parallelogramma = 7.35
```

Validazione. Esercizio svolto 8.1 del testo di teoria: $\mathbf{A} \times \mathbf{B} = (-3, 6, -3)$ con modulo $\sqrt{54} \simeq 7{,}35$, e le due verifiche di ortogonalità danno zero ✓. Il caso piano conferma il paragrafo 8.3: ruotando da (3, 4) verso (4, 3) si gira in senso *orario* (il primo vettore è più «alto» del secondo), e infatti il segno è negativo.

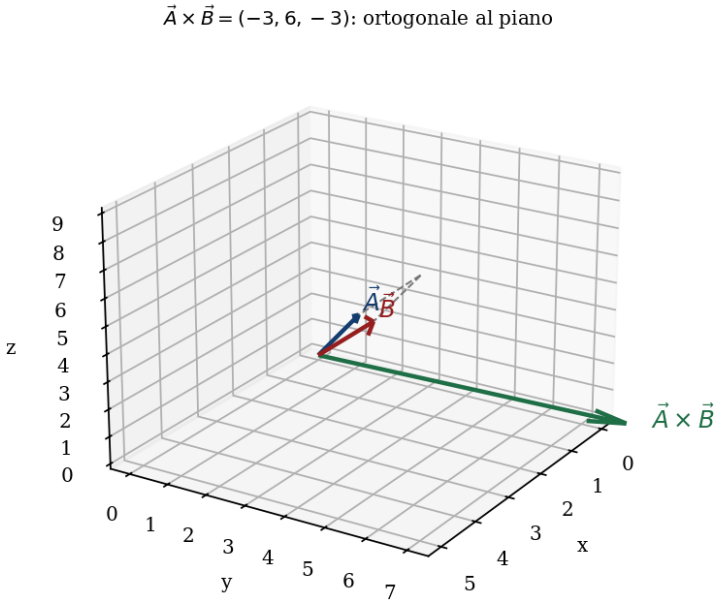


Figura 8.4. Il prodotto vettoriale nello spazio: $\mathbf{A} \times \mathbf{B}$ è perpendicolare al piano dei due fattori, e il suo modulo è l'area del parallelogramma tratteggiato. Figura generata con la proiezione 3D di Matplotlib.

Osservazione

L'ortogonalità è il collaudo perfetto del prodotto vettoriale: due prodotti scalari, due zeri, nessun valore di riferimento da cercare su un libro. Se sbagliate un segno nel determinante, uno dei due zeri smette di essere zero e l'errore si autodenuncia immediatamente. Adottate questo controllo *sempre*, anche quando siete sicuri.

8.8 Nello spazio: coseni direttori

Il paragrafo 5.4 del testo di teoria introduce i coseni direttori e la loro identità fondamentale. In Python l'osservazione decisiva è questa: *i coseni direttori sono le componenti del versore*, e quindi si ottengono con una sola divisione.

Formula chiave

$$\cos\theta_x = \frac{v_x}{v}, \quad \cos\theta_y = \frac{v_y}{v}, \quad \cos\theta_z = \frac{v_z}{v}, \qquad \cos^2\theta_x + \cos^2\theta_y + \cos^2\theta_z = 1$$

L'identità non è altro che $\|\hat{v}\| = 1$ scritta per componenti: per questo è il collaudo naturale del calcolo.

`direttori.py`

```python
# direttori.py -- coseni direttori di un vettore nello spazio
import numpy as np

v = np.array([2.0, 3.0, 6.0])
mod = np.linalg.norm(v)
cosini = v/mod                          # i coseni direttori SONO il versore
angoli = np.degrees(np.arccos(cosini))

print(f"modulo   = {mod:.0f}")
print(f"versore  = {np.round(cosini, 4)}")
for asse, c, a in zip("xyz", cosini, angoli):
    print(f"  cos(theta_{asse}) = {c:.4f}   theta_{asse} = {a:.2f} gradi")
print(f"identita' fondamentale: somma dei quadrati = {(cosini**2).sum():.6f}")
```

Output

```
modulo   = 7
versore  = [0.2857 0.4286 0.8571]
  cos(theta_x) = 0.2857   theta_x = 73.40 gradi
  cos(theta_y) = 0.4286   theta_y = 64.62 gradi
  cos(theta_z) = 0.8571   theta_z = 31.00 gradi
identita' fondamentale: somma dei quadrati = 1.000000
```

Validazione. Esercizio svolto 5.3 del testo di teoria: modulo 7, versore (2/7, 3/7, 6/7), angoli 73,40, 64,62 e 31,00 gradi ✓. La terna (2, 3, 6) è «pitagorica spaziale»: $4 + 9 + 36 = 49$. Notate il ciclo `zip("xyz", cosini, angoli)`: una stringa in Python è iterabile carattere per carattere, e questo consente di etichettare i tre assi senza scrivere tre volte la stessa riga.

8.9 Il momento di una forza e la coppia

Arriviamo all'operazione che nel corso di Strutture faremo mille volte. Il momento è un prodotto vettoriale, ma nel piano se ne usa la sola componente z: una funzione di due righe che diventerà il vostro strumento quotidiano.

Formula chiave

$$\mathbf{M}_P = \mathbf{r} \times \mathbf{F}, \qquad \mathbf{r} = \overrightarrow{PA} \qquad \Longrightarrow \qquad M_z = r_x F_y - r_y F_x$$

con la convenzione: $M_z > 0$ antiorario (uscente dal foglio), $M_z < 0$ orario (entrante). Il modulo vale $F\,b$, con b braccio.

Prompt pronto all'uso

Scrivi un programma Python per il momento di una forza nel piano. Modello: funzione momento_z(polo, punto, F) che calcoli $r_xF_y - r_yF_x$ con $\mathbf{r}$ = punto − polo, e ne dichiari il verso (antiorario se positivo). Dati: forza $(0, -2000)$ N applicata in $(0{,}5;\ 0)$ m, polo nell'origine. Poi verifica la proprietà della coppia: due forze $\pm(400, 0)$ N applicate in $(0,\ 0)$ e $(0;\ 0{,}6)$, calcolando il momento rispetto a tre poli diversi scelti a caso. Controlli: il primo momento deve valere -1000 N·m (orario) e coincidere con F b; i tre momenti della coppia devono essere identici fra loro e pari a $+240$ N·m.

momento.py

```python
# momento.py -- momento di una forza rispetto a un polo (piano)
import numpy as np

def momento_z(polo, punto, F):
    """Componente z di M = r x F, con r = punto - polo. Segno + =
    → antiorario."""
    r = np.asarray(punto) - np.asarray(polo)
    return r[0]*F[1] - r[1]*F[0]

O = (0.0, 0.0)
P = (0.5, 0.0)                  # punto di applicazione
F = np.array([0.0, -2000.0])    # forza verticale verso il basso [N]

M = momento_z(O, P, F)
print(f"M_O = {M:+.0f} N m   ->  senso {'antiorario' if M > 0 else 'orario'}")

# controllo con la formula del braccio: |M| = F * b
b = 0.5                         # forza perpendicolare al braccio
print(f"controllo |M| = F b = {np.linalg.norm(F)*b:.0f} N m")

# la coppia: risultante nulla, momento indipendente dal polo
Fc = np.array([400.0, 0.0])
A, B = (0.0, 0.0), (0.0, 0.6)
print("\nCoppia di 400 N con braccio 0.6 m:")
for polo in [(0.0, 0.0), (1.5, 2.0), (-3.0, 7.5)]:
    Mc = momento_z(polo, A, Fc) + momento_z(polo, B, -Fc)
    print(f"  polo {str(polo):12s} -> M = {Mc:+.0f} N m")
```

Output

```
M_O = -1000 N m   ->  senso orario
controllo |M| = F b = 1000 N m

Coppia di 400 N con braccio 0.6 m:
  polo (0.0, 0.0)   -> M = +240 N m
  polo (1.5, 2.0)   -> M = +240 N m
  polo (-3.0, 7.5)  -> M = +240 N m
```

Validazione. La prima parte è l'esercizio svolto 8.2 del testo di teoria ($\mathbf{M}_O = (0,\ 0,\ -1000)$, cioè 1000 N m orari) ✓; la seconda è l'esercizio 9.4, con in più un terzo polo scelto *a caso*, lontano dalla struttura. Il momento della coppia resta +240 N m: il programma non verifica un caso, verifica una *proprietà*. È la traduzione numerica della dimostrazione del paragrafo 10.5: il risultato dipende solo da $\overrightarrow{BA}$, non dal polo.

Prova tu

Il polo casuale non basta a rendere una verifica rigorosa. Riscrivete il ciclo estraendo dieci poli con `np.random.uniform(-50, 50, 2)` e controllate che la differenza massima fra i momenti sia inferiore a 10^{-9}. È il vostro primo *test automatico*: non controllate più i numeri a occhio, ma fate controllare al programma una proprietà matematica.

8.10 Il teorema di Varignon verificato al calcolatore

Il paragrafo 10.4 del testo di teoria dimostra il teorema di Varignon per forze concorrenti in tre righe, invocando la distributività del prodotto vettoriale. Verificarlo numericamente non aggiunge nulla alla dimostrazione — una dimostrazione non ha bisogno di conferme — ma insegna qualcosa di diverso: come si costruisce un collaudo che confronta due strade di calcolo.

`varignon.py`

```
# varignon.py -- verifica numerica del teorema di Varignon
import numpy as np

def momento_z(polo, punto, F):
    r = np.asarray(punto) - np.asarray(polo)
    return r[0]*F[1] - r[1]*F[0]

# tre forze CONCORRENTI nel punto A (ipotesi del teorema)
A = (1.2, 0.8)
forze = [np.array([300.0,   0.0]),
         np.array([  0.0, -250.0]),
         np.array([-80.0,  150.0])]
polo = (0.0, 0.0)

somma_momenti = sum(momento_z(polo, A, F) for F in forze)
R = sum(forze)
momento_risultante = momento_z(polo, A, R)

print(f"R = ({R[0]:.1f}, {R[1]:.1f}) N")
print(f"somma dei momenti     = {somma_momenti:+.2f} N m")
print(f"momento della risultante = {momento_risultante:+.2f} N m")
print(f"differenza = {abs(somma_momenti - momento_risultante):.2e}  ->
    → Varignon verificato")
```

Output

```
R = (220.0, -100.0) N
somma dei momenti     = -296.00 N m
momento della risultante = -296.00 N m
differenza = 0.00e+00  -> Varignon verificato
```

Lettura. Le due grandezze coincidono esattamente, e non per caso: sono la stessa espressione algebrica calcolata in ordine diverso. Il valore della verifica non sta nel numero, ma nella struttura del programma — *sommare i momenti* oppure *ridurre e poi calcolare* sono due percorsi che Varignon autorizza a scambiare. Nei problemi reali si sceglie il più corto.

Osservazione

L'ipotesi «forze concorrenti» è essenziale e nel codice si vede: tutte le forze condividono lo stesso punto di applicazione `A`, quindi lo stesso braccio $\mathbf{r}$. Se i punti di applicazione fossero diversi, l'uguaglianza varrebbe solo collocando $\mathbf{R}$ sul suo *asse centrale*, come avverte il testo di teoria. Provate a cambiare un punto di applicazione: la differenza smette di essere nulla, e avrete visto *perché* l'ipotesi serve.

8.11 Il triangolo del vento

Ecco l'applicazione aeronautica per eccellenza della somma vettoriale (paragrafo 6.4 del testo di teoria). Il programma aggiunge un elemento che il calcolo a mano rende laborioso: la scomposizione del vento in componente frontale e trasversale, cioè le due grandezze che il pilota usa davvero.

Formula chiave

$$\mathbf{V}_g = \mathbf{V}_{air} + \mathbf{V}_{wind}, \qquad V_{head} = \mathbf{V}_{wind} \cdot \hat{u}, \qquad V_{cross} = \left\| \mathbf{V}_{wind} - V_{head}\,\hat{u} \right\|$$

con $\hat{u}$ versore della prua. Le ultime due sono proiezione e componente residua: prodotto scalare all'opera.

Prompt pronto all'uso

Scrivi un programma Python sul triangolo del vento. Modello: somma vettoriale $\mathbf{V}_g = \mathbf{V}_{air} + \mathbf{V}_{wind}$; ground speed come modulo; deriva come differenza fra l'angolo della rotta e quello della prua, entrambi con atan2; scomposizione del vento in headwind e crosswind per proiezione sul versore della prua. Dati: TAS $(150, 0)$ kt (prua a Est), vento $(-30, -40)$ kt (sistema: x Est, y Nord). Formato: velocità in kt con due decimali, angoli in gradi. Controlli: la ground speed deve valere $126{,}49$ kt e la deriva $-18{,}43$ gradi; headwind e crosswind devono risultare 30 e 40 kt, e il loro comporsi deve restituire il modulo del vento (50 kt).

`vento.py`

```python
# vento.py -- il triangolo del vento con NumPy
import numpy as np, math

V_air  = np.array([150.0,   0.0])   # TAS, prua a Est [kt]
V_wind = np.array([-30.0, -40.0])   # vento [kt]

V_g = V_air + V_wind                # composizione delle velocita'
GS = np.linalg.norm(V_g)
rotta = math.degrees(math.atan2(V_g[1], V_g[0]))
prua  = math.degrees(math.atan2(V_air[1], V_air[0]))

print(f"V_g  = ({V_g[0]:.1f}, {V_g[1]:.1f}) kt")
print(f"ground speed = {GS:.2f} kt   (TAS = {np.linalg.norm(V_air):.1f} kt)")
print(f"prua = {prua:.1f} gradi   rotta = {rotta:.2f} gradi")
print(f"angolo di deriva = {rotta - prua:.2f} gradi")

# scomposizione del vento rispetto alla prua
u = V_air/np.linalg.norm(V_air)          # versore della prua
head = np.dot(V_wind, u)                 # componente lungo la prua
cross = np.linalg.norm(V_wind - head*u)  # componente trasversale
print(f"\nheadwind = {-head:.1f} kt (contraria)   crosswind = {cross:.1f} kt")
```

```
Output
```

```
V_g  = (120.0, -40.0) kt
ground speed = 126.49 kt   (TAS = 150.0 kt)
prua = 0.0 gradi   rotta = -18.43 gradi
angolo di deriva = -18.43 gradi

headwind = 30.0 kt (contraria)   crosswind = 40.0 kt
```

Validazione. Esercizio svolto 6.2 del testo di teoria: $\mathbf{V}_g = (120, -40)$ kt, $GS \simeq 126{,}49$ kt, deriva $-18{,}43$ gradi ✓. In più, headwind e crosswind valgono esattamente 30 e 40 kt: il vento è la terna pitagorica 30–40–50 allineata con gli assi, e la sua scomposizione lo dimostra ($\sqrt{30^2 + 40^2} = 50$).

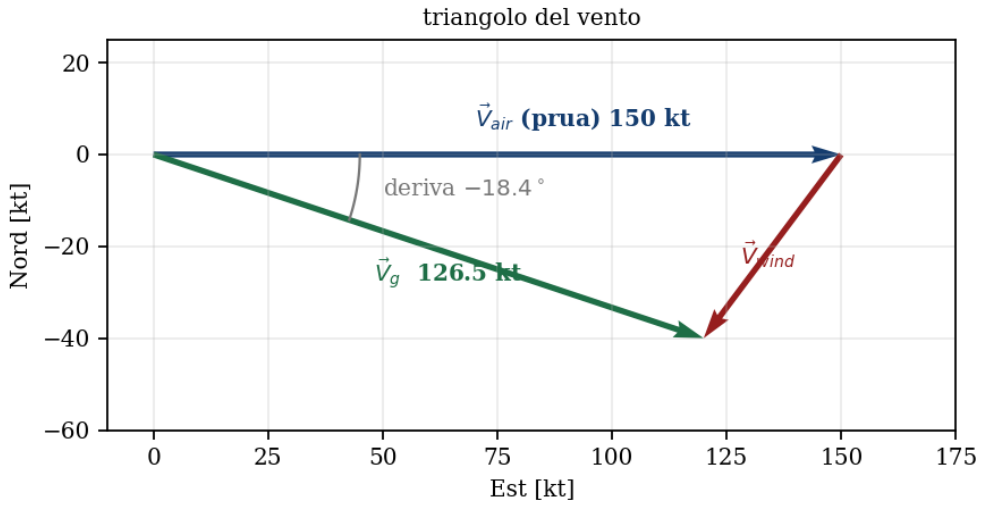


Figura 8.5. Il triangolo del vento costruito testa–coda: la prua (blu) più il vento (rosso) danno la rotta effettiva al suolo (verde). Confrontate con la Figura 9 del testo di teoria.

Prova tu

Il problema *inverso* è quello che il pilota risolve davvero: assegnata la rotta desiderata, quale prua occorre tenere per compensare la deriva? Impostatelo come ricerca della correzione δ che annulla la componente trasversale della rotta, e risolvetelo con la bisezione. Poi tabulate la prua di correzione per un vento di 40 kt proveniente da otto direzioni diverse.

8.12 Il tema d'esame: ridurre un sistema di forze

Chiudiamo con il problema che il testo di teoria affronta nel capitolo 11: un rettangolo 2×1 m con tre forze applicate ai vertici, di cui si chiedono risultante, momenti rispetto a poli diversi e riduzione a un punto. È l'esercizio in cui converge tutto, ed è anche quello che meglio mostra il vantaggio del calcolatore: la tabella dei momenti rispetto a cinque poli, che a mano richiede venti moltiplicazioni con altrettante occasioni di sbagliare un segno, qui è un ciclo di tre righe.

Prompt pronto all'uso

Scrivi un programma Python che riduca un sistema di forze piane a un polo assegnato. Modello: funzione riduci(sistema, polo) dove sistema è una lista di coppie (forza, punto di applicazione); restituisce la risultante come somma vettoriale e il momento risultante come somma dei momenti. Dati: rettangolo con vertici $A(0,1)$, $B(0,0)$, $C(2,1)$, $D(2,0)$ m e forze $\mathbf{F}_1 = (0,-250)$ N in A, $\mathbf{F}_2 = (300,0)$ N in B, $\mathbf{F}_3 = (0,150)$ N in C; polo di riduzione $Q(1,0)$. Formato: risultante, modulo, angolo, poi una tabella dei momenti di ciascuna forza rispetto a tutti e cinque i punti notevoli. Controlli: per il sistema a due forze $|\mathbf{R}| = 390{,}51$ N e $M_Q = +250$ N·m; per quello a tre $|\mathbf{R}| = 316{,}23$ N e $M_Q = +400$ N·m.

riduzione.py

```python
# riduzione.py -- riduzione di un sistema di forze a un polo
import numpy as np, math

def momento_z(polo, punto, F):
    r = np.asarray(punto) - np.asarray(polo)
    return r[0]*F[1] - r[1]*F[0]

def riduci(sistema, polo):
    """Restituisce (risultante, momento risultante) rispetto al polo."""
    R = sum(F for F, _ in sistema)
    M = sum(momento_z(polo, P, F) for F, P in sistema)
    return R, M

# geometria della traccia: rettangolo 2 m x 1 m
V = {"A": (0.0, 1.0), "B": (0.0, 0.0), "C": (2.0, 1.0),
     "D": (2.0, 0.0), "Q": (1.0, 0.0)}

sistema = [(np.array([  0.0, -250.0]), V["A"]),    # F1
           (np.array([300.0,    0.0]), V["B"]),    # F2
           (np.array([  0.0,  150.0]), V["C"])]    # F3

for nome, sub in [("Sistema a 2 forze", sistema[:2]),
                  ("Sistema a 3 forze", sistema)]:
    R, M = riduci(sub, V["Q"])
    print(f"{nome}: R = ({R[0]:.0f}, {R[1]:.0f}) N  |R| =
    → {np.linalg.norm(R):.2f} N"
          f"  alfa = {math.degrees(math.atan2(R[1], R[0])):.2f} gradi  M_Q =
    → {M:+.0f} N m")

print("\npolo    M(F1)   M(F2)   M(F3)    M_tot")
for nome, P in V.items():
    m = [momento_z(P, punto, F) for F, punto in sistema]
    print(f"  {nome}   {m[0]:+6.0f}  {m[1]:+6.0f}  {m[2]:+6.0f}
    → {sum(m):+6.0f}")
```

```
Output
```

```
Sistema a 2 forze: R = (300, -250) N  |R| = 390.51 N  alfa = -39.81 gradi  M_Q =
    +250 N m
Sistema a 3 forze: R = (300, -100) N  |R| = 316.23 N  alfa = -18.43 gradi  M_Q =
    +400 N m

polo    M(F1)   M(F2)   M(F3)    M_tot
  A       -0    +300    +300     +600
  B       -0      +0    +300     +300
  C     +500    +300      +0     +800
  D     +500      -0      +0     +500
  Q     +250      -0    +150     +400
```

Validazione. La tabella riproduce, riga per riga, quella del paragrafo 11.4 del testo di teoria: momenti totali 600, 300, 800, 500 e 400 N m rispetto ad A, B, C, D e Q ✓; e le due riduzioni danno $|\mathbf{R}| = 390{,}51$ N con $M_Q = +250$ N m e $|\mathbf{R}| = 316{,}23$ N con $M_Q = +400$ N m ✓. Venti momenti calcolati, venti valori corretti, nessun segno smarrito.

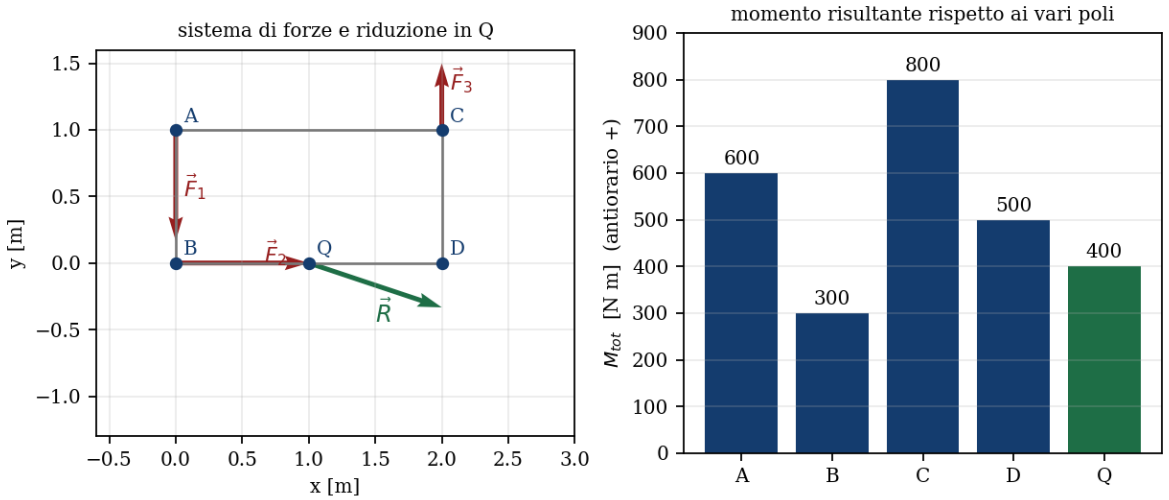


Figura 8.6. Il tema d'esame del testo di teoria: a sinistra la geometria con le tre forze e la risultante ridotta in Q; a destra il momento risultante rispetto ai cinque poli. Il momento dipende dal polo — ma in modo governato da Varignon.

Osservazione

Il grafico a destra rende evidente una proprietà che nella tabella del testo si intuisce a fatica: il momento risultante *cambia* da polo a polo, perché la risultante non è nulla. La variazione fra due poli non è però arbitraria: vale $M_{P'} - M_P = \overrightarrow{PP'} \times \mathbf{R}$. Verificatelo prendendo B e D: $\overrightarrow{BD} = (2, 0)$ e $\mathbf{R} = (300, -100)$, quindi la variazione è $2 \cdot (-100) - 0 \cdot 300 = -200$, e infatti $M_D - M_B = 500 - 300 = 200$ in modulo.

8.13 Una libreria personale: `vettori2d.py`

I dieci programmi di questo capitolo ripetono le stesse funzioni. Raccoglierle in un modulo è l'atto con cui si smette di fare esercizi e si comincia a costruire strumenti — lo stesso passo compiuto nel capitolo 4 con `atmosfera.py`.

vettori2d.py

```python
# vettori2d.py -- piccola libreria di vettori nel piano per il corso
import numpy as np, math

def polare(modulo, gradi):
    """Da modulo e angolo (gradi dall'asse x) alle componenti."""
    a = math.radians(gradi)
    return modulo*np.array([math.cos(a), math.sin(a)])

def modulo(v):
    return float(np.linalg.norm(v))

def angolo(v):
    """Angolo del vettore con l'asse x, in gradi, quadrante corretto."""
    return math.degrees(math.atan2(v[1], v[0]))

def versore(v):
    m = modulo(v)
    if m == 0.0:
        raise ValueError("il vettore nullo non ha versore")
    return v/m

def angolo_tra(u, v):
    """Angolo fra due vettori, in gradi (0..180)."""
    c = np.dot(u, v)/(modulo(u)*modulo(v))
    return math.degrees(math.acos(np.clip(c, -1.0, 1.0)))

def cross_z(u, v):
    """Componente z del prodotto vettoriale nel piano."""
    return float(u[0]*v[1] - u[1]*v[0])

def momento_z(polo, punto, F):
    """Momento di F applicata in punto, rispetto a polo. + = antiorario."""
    r = np.asarray(punto, dtype=float) - np.asarray(polo, dtype=float)
    return cross_z(r, F)

def riduci(sistema, polo):
    """sistema = [(F, punto), ...] -> (risultante, momento risultante)."""
    R = sum(np.asarray(F, dtype=float) for F, _ in sistema)
    M = sum(momento_z(polo, P, F) for F, P in sistema)
    return R, M
```

Due dettagli meritano attenzione, perché sono scelte di progetto e non di scrittura. Il primo: `versore()` solleva un'eccezione sul vettore nullo anziché restituire un risultato assurdo — il vettore nullo non ha direzione, e il programma lo dichiara invece di inventarsela. Il secondo: `np.clip(c, -1.0, 1.0)` in `angolo_tra()`.

Osservazione

Perché quel `clip`? Perché per due vettori quasi paralleli l'arrotondamento in virgola mobile può produrre $\cos\gamma = 1{,}0000000000000002$, e `math.acos` di un numero maggiore di uno solleva `ValueError`: il programma si ferma su un caso perfettamente legittimo. Il `clip` riporta il valore nell'intervallo ammissibile. È il tipo di difesa che distingue una funzione di libreria da una riga di esercizio.

prova_libreria.py

```python
# prova_libreria.py -- collaudo della libreria sugli esercizi del testo
import numpy as np
from vettori2d import polare, modulo, angolo, versore, angolo_tra, cross_z,
    → riduci

print("E1  :", modulo(np.array([6.0,-8.0])), versore(np.array([6.0,-8.0])),
      f"{angolo(np.array([6.0,-8.0])):.1f} gradi")
print("E3  :", riduci([(np.array([100.0,0.0]),(0,0)),
                       (np.array([0.0,-150.0]),(0,0)),
                       (np.array([-60.0,80.0]),(0,0))], (0,0))[0])
print("E7  :", f"{modulo(np.array([900.0,1200.0])):.0f} N",
      f"{angolo(np.array([900.0,1200.0])):.1f} gradi")
print("7.1 :", f"{angolo_tra(np.array([3.0,4.0]), np.array([4.0,3.0])):.2f}
    → gradi")
print("polare(500, 120) =", np.round(polare(500,120), 2))
```

Output

```
E1  : 10.0 [ 0.6 -0.8] -53.1 gradi
E3  : [ 40. -70.]
E7  : 1500 N 53.1 gradi
7.1 : 16.26 gradi
polare(500, 120) = [-250.    433.01]
```

Validazione. Quattro esercizi del testo di teoria collaudati in cinque righe: E1 ($\|\mathbf{A}\| = 10$, $\hat{a} = (0{,}6; -0{,}8)$, $-53{,}1$ gradi), E3 ($\mathbf{R} = (40, -70)$ N), E7 (1500 N a 53,1 gradi, terna 3–4–5 scalata $\times 300$) e l'esercizio svolto 7.1 (16,26 gradi) ✓. Una libreria che riproduce le risposte del libro è una libreria di cui ci si può fidare.

8.14 Errori comuni con i vettori in Python

Il testo di teoria dedica i riquadri rossi agli errori concettuali; qui raccogliamo quelli specifici del calcolatore, che si aggiungono ai primi e sono altrettanto insidiosi perché non producono messaggi.

1. **Gradi al posto dei radianti.** Ogni funzione trigonometrica di Python lavora in radianti. Regola operativa: convertite in ingresso con `math.radians`, calcolate in radianti, convertite in uscita con `math.degrees`. Sintomo tipico: un seno di 30 che vale $-0{,}988$ invece di 0,5.

2. **`atan` al posto di `atan2`.** Perde il quadrante e divide per zero sui vettori verticali. Non c'è mai una ragione per usare `atan` su un vettore.

3. **Confronto con lo zero esatto.** Usate `abs(x) < 1e-9`, mai `x == 0`, su risultati di calcoli in virgola mobile.

4. **Il vettore nullo.** Non ha versore né angolo. Se il codice deve gestirlo, va trattato esplicitamente.

5. **`np.cross` su vettori del piano.** Funziona, ma restituisce uno scalare (la componente z), non un array: se vi aspettate un vettore, il codice successivo si rompe. Meglio la funzione `cross_z` esplicita.

6. **Array di interi.** `np.array([3, 4])` è un array di interi: certe operazioni troncano senza avvisare. Scrivete sempre i decimali.

7. **Lo zero negativo.** Guardate la tabella del paragrafo precedente: compaiono sia `+0` sia `-0`.

Osservazione

Lo *zero negativo* merita una spiegazione, perché in aula genera sempre stupore. Lo standard IEEE 754, che governa la virgola mobile in ogni calcolatore, prevede due zeri distinti: $+0{,}0$ e $-0{,}0$. Il prodotto $-250 \times 0{,}0$ vale $-0{,}0$, e la formattazione `{:+.0f}` lo stampa fedelmente come `-0`. Non è un errore né un bug: numericamente $-0{,}0 = +0{,}0$ (il confronto `-0.0 == 0.0` restituisce `True`), ed è solo la stampa a distinguerli. Se il segno vi disturba nella tabella, sommate zero: `m + 0.0` normalizza il risultato a $+0{,}0$. Il vero insegnamento è un altro: leggere *davvero* l'output, notare l'anomalia e cercarne la causa — invece di ignorarla — è esattamente il comportamento che il ciclo GLVC del capitolo 7 vuole costruire in voi.

8.15 Banco di prova: gli esercizi del testo di teoria

Il capitolo 12 del testo di teoria propone sedici esercizi con le risposte. Sono il banco di collaudo ideale della vostra `vettori2d.py`: le risposte esistono, quindi ogni scostamento è un errore *vostro* da cercare, non un dubbio da lasciare irrisolto.

Prova tu

Livello base — una riga di libreria ciascuno. Risolvete E1, E2, E3, E5, E6, E7 e confrontate con le risposte del testo. Suggerimento per E2: nello spazio servono `np.cross` e la verifica di ortogonalità.
Livello intermedio — navigazione e cinematica. E9 (spostamento fra waypoint: attesi 50 NM e 12 min), E10 (ground speed $\simeq 182{,}5$ kt, deriva $-9{,}5$ gradi), E11 (virata: $a_c \simeq 5{,}66\ \text{m/s}^2$, $R \simeq 635{,}6$ m, $n \simeq 1{,}15$), E12 (velocità media 50 m/s = 180 km/h).
Livello avanzato — riduzione di sistemi. E13 ($\mathbf{R} \simeq 943{,}4$ N a $-58{,}0$ gradi, $M_O = -580$ N m), E14 (variazione di momento fra due poli), E15 (coppia di trasporto 900 N m), E16 ($\mathbf{R} \simeq 447{,}2$ N a 26,6 gradi, $M_O = -1700$ N m). Risolveteli *tutti con la stessa funzione* `riduci()`: se dovete modificarla per farla funzionare su un esercizio, la sua progettazione era incompleta.
Livello grafico. Per E13 e E16 disegnate con `quiver` il sistema originale e, accanto, la riduzione al polo (risultante più arco della coppia): sono le figure che vi chiederanno di allegare alla relazione di calcolo.

Dialoga con l'IA

Chiedete all'assistente: *"Aggiungi a vettori2d.py una funzione `disegna_sistema(sistema, polo)` che tracci con Matplotlib le forze applicate nei rispettivi punti, la risultante nel polo e un arco orientato che rappresenti il momento risultante"*. Poi collaudatela sull'esercizio 9.1 del testo di teoria, i tre tiranti a 120 gradi: il disegno deve mostrare tre frecce simmetriche e *nessuna* risultante. Se compare una freccia residua, o il codice sbaglia, oppure — più probabilmente — sta disegnando un vettore di lunghezza 10^{-14} che va trattato come nullo. Riconoscere quale delle due sia il caso è precisamente ciò che questo corso vi insegna.

Parte II

Terzo anno

L'atmosfera, l'aerostatica, i vettori e il linguaggio dell'aerodinamica

Capitolo 9

Dieci problemi risolti per il terzo anno

Nota

Dieci problemi completamente risolti al calcolatore sugli argomenti del terzo anno: *unità di misura aeronautiche*, *atmosfera standard* (troposfera e bassa stratosfera), *aerostatica*, *teoria dei vettori*, *il linguaggio dell'aerodinamica*, *geometria dell'ala e dei profili*. Ogni scheda contiene: il problema, i richiami teorici, un *prompt pronto all'uso* costruito con le cinque componenti del capitolo 7, il codice collaudato, l'output autentico e la validazione manuale. Il prompt non è un optional: è l'atto di progetto; il codice che segue è ciò che dovete *ottenere e riconoscere corretto* al termine del ciclo GLVC.

9.1 Scheda 9.1 Il traduttore di unità aeronautiche

Problema. In aviazione convivono nodi, piedi, chilometri orari e metri al secondo. Costruire un traduttore che converta un elenco di grandezze operative tipiche: crociera a 250 kt, livello di volo FL350 (35 000 ft), vento al suolo di 18 kt, quota di circuito di 1000 ft.

Formula chiave

Definizioni esatte: $1\,\text{kt} = \frac{1852}{3600}\ \text{m/s} = 0{,}514\,444\ \text{m/s}$; $1\,\text{ft} = 0{,}3048\ \text{m}$ (esatto per convenzione ICAO).

Prompt pronto all'uso

Scrivi un programma Python che converta grandezze aeronautiche. Modello: fattori esatti 1 kt = 0,514444 m/s e 1 ft = 0,3048 m. Dati: una lista di tuple (descrizione, valore, unità) con almeno velocità in kt e quote in ft. Dominio: solo unità "kt" e "ft", altre unità vanno segnalate come non gestite. Formato: tabella allineata con f-string, velocità sia in m/s sia in km/h. Controllo per il collaudo: 250 kt devono dare 128,61 m/s e 35000 ft devono dare 10668,0 m.

conversioni.py

```python
# conversioni.py -- traduttore di unita' aeronautiche
KT = 0.514444          # 1 nodo in m/s (1852/3600)
FT = 0.3048            # 1 piede in m (esatto, ICAO)

dati = [("Velocita' di crociera", 250,   "kt"),
        ("Livello di volo FL350", 35000, "ft"),
        ("Vento al suolo",        18,    "kt"),
        ("Quota di circuito",     1000,  "ft")]

for nome, v, u in dati:
    if u == "kt":
        print(f"{nome:24s}: {v:6.0f} kt = {v*KT:7.2f} m/s "
              f"= {v*KT*3.6:6.1f} km/h")
    elif u == "ft":
        print(f"{nome:24s}: {v:6.0f} ft = {v*FT:7.1f} m")
    else:
```

```
        print(f"{nome:24s}: unita' '{u}' non gestita")
```

Output

```
Velocita' di crociera    :    250 kt =  128.61 m/s =  463.0 km/h
Livello di volo FL350    :  35000 ft = 10668.0 m
Vento al suolo           :     18 kt =    9.26 m/s =   33.3 km/h
Quota di circuito        :   1000 ft =   304.8 m
```

Validazione. $250 \cdot 1852/3600 = 463000/3600 = 128{,}61$ m/s ✓; e $463000/3600 \cdot 3{,}6 = 463$ km/h: i nodi in km/h si ottengono moltiplicando per 1,852 — regola mnemonica dei piloti che il programma riscopre. FL350: $35000 \cdot 0{,}3048 = 10\,668$ m ✓, appena sotto la tropopausa.

Prova tu

Aggiungete miglia nautiche (1 NM = 1852 m) e la conversione inversa m/s → kt. Poi rendete il programma interattivo con `input()`: valore e unità su una riga sola, separati da uno spazio (suggerimento: `.split()`).

9.2 Scheda 9.2 L'atmosfera standard a una quota qualsiasi

Problema. Interrogare il modello ISA a una quota scelta dall'utente (qui 7500 m) ottenendo T, p, ρ e il rapporto di densità σ, con controllo del dominio di validità.

Richiami minimi. Le formule della troposfera sono nella formula chiave del capitolo 4; $\sigma = \rho/\rho_0$ comparirà in tutte le formule di prestazione del quarto anno.

Prompt pronto all'uso

Scrivi un programma Python per l'atmosfera ISA. Modello: troposfera con $T = T_0 - \lambda h$ e $p = p_0(T/T_0)^{g/(R\lambda)}$, gas perfetto per la densità; costanti $T_0 = 288{,}15$ K, $p_0 = 101325$ Pa, $\lambda = 0{,}0065$ K/m, $R = 287{,}05$. Dati: quota in metri da input. Dominio: $0 \le h \le 11000$ m, fuori intervallo stampa un rifiuto motivato. Formato: T in K e °C, p in Pa e hPa, ρ e σ con quattro decimali. Controllo: a 7500 m deve risultare $T = 239{,}40$ K e $p = 38251$ Pa.

quota_isa.py

```
# quota_isa.py -- caratteristiche ISA a una quota richiesta
T0, p0, lam, g, R = 288.15, 101325.0, 0.0065, 9.80665, 287.05
h = float(input("Quota [m]: "))
T = T0 - lam*h
p = p0*(T/T0)**(g/(R*lam))
rho = p/(R*T)
print(f"T   = {T:.2f} K ({T-273.15:.2f} gradi C)")
print(f"p   = {p:.0f} Pa = {p/100:.1f} hPa")
print(f"rho = {rho:.4f} kg/m^3   (rho/rho0 = {rho/1.225:.4f})")
```

```
Output

Quota [m]: 7500
T   = 239.40 K (-33.75 gradi C)
p   = 38251 Pa = 382.5 hPa
rho = 0.5566 kg/m^3   (rho/rho0 = 0.4544)
```

Validazione. A 7500 m: $T = 288{,}15 - 48{,}75 = 239{,}40$ K ✓. Il rapporto $\sigma = 0{,}4544$ dice che a quella quota l'aria ha meno di metà della densità del livello del mare: a parità di C_L e di peso, la velocità vera dovrà crescere del fattore $1/\sqrt{\sigma} = 1{,}484$.

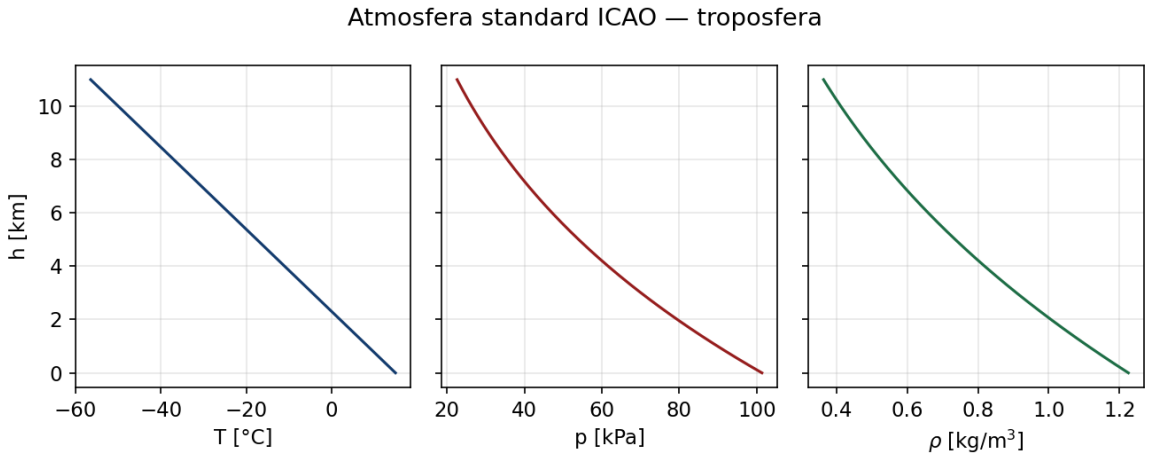


Figura 9.1. Andamenti di T, p e ρ nella troposfera, generati con Matplotlib dalla vostra `atmosfera.py`.

Prova tu

Aggiungete il controllo di dominio richiesto dal prompt (il codice qui sopra, volutamente, non lo implementa ancora: è il vostro primo collaudo negativo — provate $h = 15\,000$ e osservate il numero privo di senso fisico che esce).

9.3 Scheda 9.3 L'atmosfera fino a 20 km: entra la stratosfera

Problema. Estendere il modello ISA alla bassa stratosfera ($11000 < h \le 20\,000$ m, temperatura costante $T = 216{,}65$ K) e tabulare T, p, ρ ogni 2000 m da 0 a 20 000 m.

Formula chiave

In uno strato isotermo l'equilibrio idrostatico $\mathrm{d}p = -\rho g\,\mathrm{d}h$ con $\rho = p/(RT)$ si integra in forma esponenziale:

$$p(h) = p_{11}\,\exp\!\left[-\frac{g\,(h-11000)}{R\,T_s}\right], \qquad T_s = 216{,}65\ \mathrm{K}$$

dove $p_{11} = 22\,632$ Pa è la pressione alla tropopausa, calcolata con la formula troposferica. La dimostrazione richiede solo la separazione delle variabili: $\mathrm{d}p/p = -g/(RT_s)\,\mathrm{d}h$.

Prompt pronto all'uso

Estendi la funzione isa(h) alla stratosfera. Modello: sotto gli 11000 m formula troposferica; sopra, T costante a 216,65 K e pressione esponenziale $p = p_{11}\exp[-g(h-11000)/(RT)]$ con p_{11} calcolato dalla formula troposferica a 11000 m, non scritto a mano. Dati: nessun input, tabula da 0 a 20000 m a passo 2000. Dominio: rifiuta $h > 20000$. Formato: colonne allineate. Controlli: a 12000 m $p \simeq 19330$ Pa; a 20000 m $p \simeq 5475$ Pa; la tabella non deve avere salti alla tropopausa.

atmosfera20.py

```python
# atmosfera20.py -- ISA con troposfera e bassa stratosfera
import math
T0, p0, lam, g, R = 288.15, 101325.0, 0.0065, 9.80665, 287.05

def isa(h):
    """ISA fino a 20 km: restituisce (T, p, rho)."""
    if h <= 11000:
        T = T0 - lam*h
        p = p0*(T/T0)**(g/(R*lam))
    else:
        T = 216.65                              # strato isotermo
        p11 = p0*(216.65/T0)**(g/(R*lam))       # raccordo continuo
        p = p11*math.exp(-g*(h - 11000)/(R*T))
    return T, p, p/(R*T)

print("  h[m]     T[K]      p[Pa]    rho[kg/m^3]")
for h in range(0, 20001, 2000):
    T, p, rho = isa(h)
    print(f"{h:6d}   {T:7.2f}  {p:9.1f}    {rho:.4f}")
```

Output

```
  h[m]     T[K]      p[Pa]    rho[kg/m^3]
     0   288.15  101325.0    1.2250
  2000   275.15   79495.0    1.0065
  4000   262.15   61639.9    0.8191
  6000   249.15   47180.6    0.6597
  8000   236.15   35599.4    0.5252
 10000   223.15   26435.9    0.4127
 12000   216.65   19330.1    0.3108
 14000   216.65   14101.5    0.2268
 16000   216.65   10287.2    0.1654
 18000   216.65    7504.6    0.1207
 20000   216.65    5474.7    0.0880
```

Validazione. A 20 000 m: l'esponente vale $-9{,}80665 \cdot 9000/(287{,}05 \cdot 216{,}65) = -1{,}4195$, quindi $p = 22632 \cdot e^{-1{,}4195} = 22632 \cdot 0{,}2419 = 5475$ Pa ✓. Controllo strutturale: la colonna T è continua alla tropopausa (223,15 → 216,65 senza salti anomali) e la pressione decresce *sempre*: due proprietà che ogni tabella atmosferica deve esibire, e che costituiscono un collaudo gratuito.

Prova tu

Tracciate $p(h)$ da 0 a 20 km con Matplotlib in scala semilogaritmica (`plt.semilogy`): nella stratosfera la curva diventa una retta. Spiegate perché, partendo dalla forma esponenziale della formula chiave.

9.4 Scheda 9.4 La quota di un σ assegnato: ricerca per bisezione

Problema. A quale quota la densità dell'aria si dimezza ($\sigma = 0{,}5$)? Il problema *inverso* dell'ISA non ha formula diretta comoda: introduciamo il primo algoritmo numerico del corso, la bisezione.

Formula chiave

Se f è continua e monotona su $[a, b]$ con la soluzione all'interno, dimezzando l'intervallo dalla parte giusta l'errore massimo dopo n passi vale $(b - a)/2^n$: con $n = 40$ su 11 000 m l'incertezza scende sotto 10^{-8} m. La convergenza è *garantita*: è la forza del metodo.

Prompt pronto all'uso

Scrivi un programma Python che trovi la quota alla quale $\sigma = \rho/\rho_0$ assume un valore assegnato. Modello: $\sigma(h)$ dalla funzione isa(h) già scritta (riusala, non riscriverla); ricerca per bisezione sull'intervallo [0, 11000] m, sfruttando la monotonia di σ. Dati: sigma obiettivo 0,5. Dominio: accetta solo $0{,}29 < \sigma < 1$ (limiti troposferici). Formato: quota con un decimale e verifica finale di σ alla quota trovata. Controllo: il risultato atteso è poco sotto i 6700 m.

`quota_sigma.py`

```python
# quota_sigma.py -- quota per un rapporto di densita' assegnato
from atmosfera20 import isa

def sigma(h):
    T, p, rho = isa(h)
    return rho/1.225

obiettivo = 0.5
a, b = 0.0, 11000.0            # sigma(a)=1 > obiettivo > sigma(b)
for _ in range(40):            # 40 dimezzamenti: errore < 1e-8 m
    m = (a + b)/2
    if sigma(m) > obiettivo:   # densita' ancora troppo alta:
        a = m                  #   la quota cercata e' piu' su
    else:
        b = m

h = (a + b)/2
print(f"sigma = {obiettivo}  alla quota h = {h:.1f} m")
print(f"verifica: sigma({h:.1f}) = {sigma(h):.6f}")
```

Output

```
sigma = 0.5  alla quota h = 6662.8 m
verifica: sigma(6662.8) = 0.500000
```

Validazione (analitica, per questa volta possibile). Nella troposfera $\sigma = (1 - \lambda h/T_0)^{g/(R\lambda)-1}$ con esponente $5{,}2559 - 1 = 4{,}2559$; imponendo $\sigma = 0{,}5$: $h = \frac{T_0}{\lambda}\left(1 - 0{,}5^{1/4{,}2559}\right) = 44330{,}8 \cdot (1 - 0{,}8497) = 6663$ m ✓.

La bisezione riproduce la soluzione chiusa a meno del decimetro: da ora in poi potrete fidarvene anche dove la soluzione chiusa *non esiste*.

Osservazione

Notare `from atmosfera20 import isa`: il riuso del modulo validato nella scheda precedente. Non si riscrive ciò che è già collaudato — principio di ingegneria del software e, insieme, di igiene mentale.

Prova tu

Con lo stesso schema trovate la quota alla quale $p = 500$ hPa (la "quota di metà pressione" dei meteorologi) e confrontatela con quella di metà densità: perché non coincidono? La risposta è nella temperatura.

9.5 Scheda 9.5 Aerostatica: il carico utile di un pallone

Problema. Un pallone a elio da $V = 500\ \mathrm{m}^3$ con struttura (involucro e cesto) da 120 kg opera al livello del mare. Quanto carico utile solleva?

Formula chiave

Spinta aerostatica: $F_A = \rho_{aria}\, V\, g$. Il carico utile sollevabile è

$$m_u = \frac{F_A - \rho_{gas} V g - m_s g}{g} = (\rho_{aria} - \rho_{gas})\, V - m_s$$

con m_s massa della struttura (involucro e cesto).

Prompt pronto all'uso

Scrivi un programma Python sull'aerostatica. Modello: principio di Archimede applicato all'aria, bilancio di forze verticali spinta − peso gas − peso struttura. Dati: volume 500 m³, elio ρ = 0,1786 kg/m³, aria ISA al mare 1,225 kg/m³, struttura 120 kg. Dominio: risultato negativo = pallone che non decolla, va segnalato. Formato: le tre forze in N e il carico utile in kg. Controllo: il carico utile deve valere 403,2 kg, ottenibile anche dalla forma compatta $(\rho_a - \rho_{He})V - m_s$.

`pallone.py`

```
# pallone.py -- spinta aerostatica di un pallone ad elio
g = 9.80665
rho_aria = 1.225       # [kg/m^3] aria al livello del mare
rho_He   = 0.1786      # [kg/m^3] elio
V        = 500.0       # [m^3] volume del pallone
m_strutt = 120.0       # [kg] involucro + cesto

spinta   = rho_aria * V * g              # principio di Archimede
peso_gas = rho_He * V * g
peso_tot = peso_gas + m_strutt * g
carico   = (spinta - peso_tot) / g       # carico utile sollevabile

print(f"Spinta di Archimede : {spinta:8.0f} N")
print(f"Peso dell'elio      : {peso_gas:8.0f} N")
print(f"Peso della struttura: {m_strutt*g:8.0f} N")
```

```
print(f"Carico utile        : {carico:8.1f} kg")
```

Output

```
Spinta di Archimede :     6007 N
Peso dell'elio      :      876 N
Peso della struttura:     1177 N
Carico utile        :    403.2 kg
```

Validazione. Con la forma compatta: $m_u = (1{,}225 - 0{,}1786) \cdot 500 - 120 = 523{,}2 - 120 = 403{,}2$ kg ✓. Si noti la coerenza fra le due strade (bilancio delle forze e formula compatta): quando due derivazioni indipendenti coincidono, la fiducia nel risultato è dimostrata, non sperata.

Prova tu

Il pallone sale: la densità dell'aria cala secondo l'ISA mentre (in prima approssimazione) il pallone mantiene volume e gas. Con un ciclo sulla quota, trovate la *quota di tangenza statica*: la prima quota alla quale il carico utile residuo (con $m_u = 200$ kg a bordo) si annulla.

9.6 Scheda 9.6 Vettori: velocità al suolo con vento al traverso

Problema. Un velivolo vola con prua Nord a $TAS = 60$ m/s; il vento soffia da Ovest a 12 m/s. Determinare velocità al suolo e angolo di deriva.

Richiami minimi. La velocità al suolo è la somma vettoriale $\mathbf{V}_{GS} = \mathbf{V}_{TAS} + \mathbf{V}_w$. Si lavora per componenti (Est, Nord); modulo con Pitagora, direzione con l'arcotangente.

Prompt pronto all'uso

Scrivi un programma Python di navigazione. Modello: somma vettoriale per componenti Est/Nord, modulo con Pitagora, angolo con math.atan2 (non atan: motiva la scelta in un commento). Dati: TAS 60 m/s con prua 0° (Nord), vento da Ovest 12 m/s. Dominio: angoli in gradi da Nord in senso orario, conversione con math.radians. Formato: GS con due decimali, deriva in gradi. Controllo: GS deve valere $\sqrt{60^2 + 12^2} = 61{,}19$ *m/s e la deriva* $\arctan(0{,}2) = 11{,}31°$.

vento.py

```
# vento.py -- velocita' al suolo con vento al traverso
import math

TAS   = 60.0      # velocita' vera [m/s]
rotta = 0.0       # prua verso Nord [gradi]
Vw    = 12.0      # vento da Ovest verso Est [m/s]

vx = TAS * math.sin(math.radians(rotta)) + Vw   # componente Est
vy = TAS * math.cos(math.radians(rotta))        # componente Nord

GS     = math.sqrt(vx**2 + vy**2)
deriva = math.degrees(math.atan2(vx, vy))

print(f"Ground speed GS = {GS:.2f} m/s")
print(f"Angolo di deriva = {deriva:.2f} gradi")
```

Output

```
Ground speed GS = 61.19 m/s
Angolo di deriva = 11.31 gradi
```

Validazione. $GS = \sqrt{60^2 + 12^2} = \sqrt{3744} = 61{,}19$ m/s; $\delta = \arctan(12/60) = \arctan 0{,}2 = 11{,}31°$ ✓.

Osservazione

`math.atan2(vx, vy)` restituisce l'angolo nel quadrante corretto anche quando vy è negativo o nullo, dove il semplice `atan(vx/vy)` fallirebbe (divisione per zero) o sbaglierebbe quadrante. È il tipico dettaglio che distingue un programma da esercizio da uno affidabile.

Prova tu

Generalizzate a rotta e direzione del vento qualsiasi (entrambe da input, in gradi da Nord) e verificate i casi limite: vento in coda puro ($GS = TAS + V_w$, deriva nulla) e vento frontale puro.

9.7 Scheda 9.7 Il tubo di Pitot: dalla pressione alla velocità

Problema. L'anemometro misura $\Delta p = p_t - p_s$; ricavare la velocità per una serie di valori di Δp da 200 a 2000 Pa, in aria ISA al livello del mare.

Formula chiave

Per flusso incomprimibile, dal teorema di Bernoulli fra corrente indisturbata e punto di ristagno: $p_t - p_s = \frac{1}{2}\rho V^2$, da cui

$$V = \sqrt{\frac{2\,(p_t - p_s)}{\rho}}$$

Approssimazione lecita per $M \lesssim 0{,}3$; oltre, la comprimibilità richiede la correzione che vedrete al quinto anno.

Prompt pronto all'uso

Scrivi un programma Python per il tubo di Pitot. Modello: Bernoulli incomprimibile invertito, $V = \sqrt{2\Delta p/\rho}$ con $\rho = 1{,}225$ kg/m³. Dati: lista di Δp = 200, 500, 1000, 1531, 2000 Pa. Dominio: valido per $M < 0{,}3$, cioè $V < 102$ m/s circa; segnala se un valore lo supera. Formato: tabella $\Delta p \to V$ in m/s e km/h. Controllo: $\Delta p = 1531$ Pa deve restituire esattamente $V = 50{,}00$ m/s (inversione dell'esempio della pressione dinamica del capitolo 2).

pitot.py

```python
# pitot.py -- velocita' dal tubo di Pitot (flusso incomprimibile)
import math
rho = 1.225
dp_list = [200, 500, 1000, 1531, 2000]   # p_totale - p_statica [Pa]
for dp in dp_list:
    V = math.sqrt(2 * dp / rho)
    print(f"dp = {dp:5d} Pa  ->  V = {V:6.2f} m/s = {V*3.6:6.1f} km/h")
```

Output

```
dp =   200 Pa  ->  V =  18.07 m/s =   65.1 km/h
dp =   500 Pa  ->  V =  28.57 m/s =  102.9 km/h
dp =  1000 Pa  ->  V =  40.41 m/s =  145.5 km/h
dp =  1531 Pa  ->  V =  50.00 m/s =  180.0 km/h
dp =  2000 Pa  ->  V =  57.14 m/s =  205.7 km/h
```

Validazione. La riga $\Delta p = 1531\,\text{Pa} \to V = 50{,}00\,\text{m/s}$ è l'inversione esatta dell'esempio del capitolo 2 ($q(50\,\text{m/s}) = 1531{,}25\,\text{Pa}$): il cerchio si chiude, e chiudere i cerchi è il modo più elegante di validare.

Prova tu

L'anemometro è tarato con ρ_0: in quota indica quindi la *velocità indicata* V_{IAS}, non quella vera. Scrivete un programma che, date Δp e la quota, stampi V_{IAS} (calcolata con ρ_0) e V_{TAS} (con la ρ ISA della quota, importando `atmosfera20.py`) e il loro rapporto $1/\sqrt{\sigma}$.

9.8 Scheda 9.8 Reynolds e Mach al variare della quota

Problema. Un velivolo con corda media $c = 1{,}5$ m vola a $V = 50$ m/s (velocità vera) a 0, 3000, 6000 e 9000 m. Calcolare numero di Reynolds e numero di Mach a ciascuna quota.

Formula chiave

$$Re = \frac{\rho V c}{\mu}, \qquad M = \frac{V}{a}, \quad a = \sqrt{\gamma R T}, \qquad \mu(T) = \frac{1{,}458 \cdot 10^{-6}\, T^{3/2}}{T + 110{,}4} \ \text{(Sutherland)}$$

La viscosità dipende *solo* dalla temperatura: in quota cala poco, mentre ρ crolla; per questo Re diminuisce salendo.

Prompt pronto all'uso

Scrivi un programma Python che calcoli Reynolds e Mach in quota. Modello: atmosfera dalla funzione isa(h) del modulo atmosfera20 (riusala); viscosità con la legge di Sutherland $\mu = 1{,}458 \cdot 10^{-6}\, T^{1{,}5}/(T + 110{,}4)$; velocità del suono $a = \sqrt{\gamma R T}$ con $\gamma = 1{,}4$. Dati: $V = 50$ m/s, $c = 1{,}5$ m, quote 0, 3000, 6000, 9000 m. Formato: μ in notazione scientifica, Re con tre cifre significative, M con quattro decimali. Controllo: al livello del mare $\mu = 1{,}789 \cdot 10^{-5}$ Pa·s e $Re = 5{,}13 \cdot 10^{6}$.

reynolds_mach.py

```python
# reynolds_mach.py -- Re e M al variare della quota
import math
from atmosfera20 import isa

V, c, gamma, R = 50.0, 1.5, 1.4, 287.05

print("  h[m]     mu[Pa s]      Re          a[m/s]     M")
for h in [0, 3000, 6000, 9000]:
    T, p, rho = isa(h)
    mu = 1.458e-6 * T**1.5 / (T + 110.4)     # Sutherland
    Re = rho * V * c / mu
    a  = math.sqrt(gamma * R * T)
    print(f"{h:6d}   {mu:.4e}  {Re:.3e}   {a:6.2f}  {V/a:.4f}")
```

Output

```
h[m]    mu[Pa s]     Re          a[m/s]    M
    0   1.7894e-05  5.135e+06   340.29  0.1469
 3000   1.6937e-05  4.026e+06   328.58  0.1522
 6000   1.5947e-05  3.103e+06   316.43  0.1580
 9000   1.4922e-05  2.344e+06   303.79  0.1646
```

Validazione. Al mare Sutherland restituisce $\mu = 1{,}7894 \times 10^{-5}$ Pa s, il valore tabellato ISA ✓; e $Re = 1{,}225 \cdot 50 \cdot 1{,}5 / 1{,}7894 \cdot 10^{-5} = 5{,}13 \cdot 10^{6}$ ✓. Lettura fisica della tabella: salendo, Re si dimezza abbondantemente (lo strato limite "sente" un fluido più viscoso in senso cinematico) mentre M *cresce* pur a velocità vera costante, perché l'aria fredda ha velocità del suono minore: due effetti opposti della stessa salita.

Prova tu

A quale quota il velivolo, sempre a 50 m/s veri, raggiungerebbe $M = 0{,}17$? Usate la bisezione della scheda 9.4: gli algoritmi validati si riusano come mattoni.

9.9 Scheda 9.9 Geometria dell'ala trapezia

Problema. Un'ala trapezia ha apertura $b = 10$ m, corda di radice $c_r = 1{,}8$ m e di estremità $c_t = 0{,}9$ m. Calcolare superficie, allungamento, rapporto di rastremazione, corda media aerodinamica (MAC) e sua posizione in apertura.

Formula chiave

Per l'ala trapezia, con $\lambda_r = c_t / c_r$:

$$S = \frac{c_r + c_t}{2}\, b, \qquad AR = \frac{b^2}{S}, \qquad \bar{c} = \frac{2}{3}\, c_r\, \frac{1 + \lambda_r + \lambda_r^2}{1 + \lambda_r}, \qquad y_{\bar{c}} = \frac{b}{6}\, \frac{1 + 2\lambda_r}{1 + \lambda_r}$$

Le formule di $\bar{c}$ e $y_{\bar{c}}$ discendono dall'integrale $\bar{c} = \frac{2}{S}\int_0^{b/2} c^2(y)\,\mathrm{d}y$ con $c(y)$ lineare: un buon esercizio di calcolo da svolgere una volta a mano.

Prompt pronto all'uso

Scrivi un programma Python per la geometria dell'ala trapezia. Modello: formule chiuse di S, AR, λ_r, MAC e y_{MAC} per corda linearmente variabile. Dati: $b = 10$ m, $c_r = 1{,}8$ m, $c_t = 0{,}9$ m. Dominio: richiedi $0 < c_t \le c_r$. Formato: tutte le grandezze con tre decimali e unità. Controllo doppio: MAC anche per integrazione numerica di $\frac{2}{S}\int c^2\,dy$ con np.trapezoid su 200 punti — i due valori devono coincidere entro lo 0,01%.

ala_trapezia.py

```python
# ala_trapezia.py -- geometria dell'ala trapezia
import numpy as np

b, cr, ct = 10.0, 1.8, 0.9
lam_r = ct/cr
S   = (cr + ct)/2 * b
AR = b**2 / S
mac   = 2/3 * cr * (1 + lam_r + lam_r**2)/(1 + lam_r)
ymac = b/6 * (1 + 2*lam_r)/(1 + lam_r)
```

```python

# controllo indipendente: MAC per integrazione numerica
y = np.linspace(0, b/2, 200)
c = cr + (ct - cr) * y/(b/2)          # corda lineare in apertura
mac_num = 2/S * np.trapezoid(c**2, y)

print(f"Rastremazione lambda = {lam_r:.3f}")
print(f"Superficie   S  = {S:.3f} m^2")
print(f"Allungamento AR = {AR:.3f}")
print(f"MAC (formula)   = {mac:.4f} m   (numerica: {mac_num:.4f} m)")
print(f"Posizione y_MAC = {ymac:.3f} m")
```

Output

```
Rastremazione lambda = 0.500
Superficie   S  = 13.500 m^2
Allungamento AR = 7.407
MAC (formula)   = 1.4000 m   (numerica: 1.4000 m)
Posizione y_MAC = 2.222 m
```

Validazione. $\bar{c} = \frac{2}{3} \cdot 1{,}8 \cdot \frac{1{,}75}{1{,}5} = 1{,}2 \cdot \frac{7}{6} = 1{,}400$ m esatto ✓ — e l'integrazione numerica, strada del tutto indipendente, coincide alla quarta cifra. Notare che $\bar{c} > c_m = S/b = 1{,}35$ m: la corda media *aerodinamica* pesa i quadrati delle corde e premia la radice. Su $\bar{c}$ e $y_{\bar{c}}$ si posizionerà il baricentro e il fuoco nel prosieguo del corso.

Prova tu

Trasformate il programma in funzione `def geometria(b, cr, ct):` che restituisca le cinque grandezze, e tabulate MAC e AR per λ_r da 1,0 (ala rettangolare) a 0,2: per l'ala rettangolare devono risultare $\bar{c} = c_r$ e $y_{\bar{c}} = b/4$, i vostri casi limite di collaudo.

9.10 Scheda 9.10 La geometria dei profili NACA a 4 cifre

Problema. Tradurre in codice la legge di spessore NACA a 4 cifre per un profilo della serie "12" (es. NACA 2412) e verificare che spessore massimo e sua posizione siano quelli dichiarati dalla sigla.

Formula chiave

Per un profilo NACA a 4 cifre con spessore relativo massimo t (le ultime due cifre, in percento della corda):

$$\frac{y_t}{c} = 5t\left(0{,}2969\sqrt{\bar{x}} - 0{,}1260\,\bar{x} - 0{,}3516\,\bar{x}^2 + 0{,}2843\,\bar{x}^3 - 0{,}1015\,\bar{x}^4\right), \qquad \bar{x} = \frac{x}{c}$$

Prompt pronto all'uso

Scrivi un programma Python con NumPy per il profilo NACA a 4 cifre. Modello: legge di spessore NACA 1933 con i cinque coefficienti storici (0,2969; −0,1260; −0,3516; 0,2843; −0,1015). Dati: $t = 0{,}12$, ascisse x/c da 0 a 1. Formato: tabella $x/c \to y_t/c$ a quattro decimali, poi spessore massimo $2y_t$ e sua posizione. Controllo: per $t = 0{,}12$ lo spessore massimo deve risultare 0,1200 c al 30% della corda — è la promessa della sigla, il programma deve mantenerla.

naca4.py

```python
# naca4.py -- spessore di un profilo NACA a 4 cifre
import numpy as np

t = 0.12    # spessore massimo relativo (NACA 2412 -> "12")
x = np.array([0.0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.6, 0.8, 1.0])
yt = 5*t*(0.2969*np.sqrt(x) - 0.1260*x - 0.3516*x**2
          + 0.2843*x**3 - 0.1015*x**4)

for xi, yi in zip(x, yt):
    print(f"x/c = {xi:4.2f}   y_t/c = {yi:7.4f}")
print(f"\nSpessore massimo 2*y_t = {2*yt.max():.4f} c "
      f"in x/c = {x[yt.argmax()]:.2f}")
```

Output

```
x/c = 0.00   y_t/c =  0.0000
x/c = 0.05   y_t/c =  0.0355
x/c = 0.10   y_t/c =  0.0468
x/c = 0.20   y_t/c =  0.0574
x/c = 0.30   y_t/c =  0.0600
x/c = 0.40   y_t/c =  0.0580
x/c = 0.60   y_t/c =  0.0456
x/c = 0.80   y_t/c =  0.0262
x/c = 1.00   y_t/c =  0.0013
```

Validazione. Lo spessore massimo risulta 0,1200 c al 30% della corda: esattamente ciò che promette la sigla "12" e la posizione canonica del massimo spessore dei NACA a 4 cifre. La formula, scritta nel 1933, si autodenuncia corretta.

Dialoga con l'IA

Chiedete: *"Aggiungi la linea media del NACA 2412 (curvatura 2% al 40% della corda) e disegna dorso e ventre con Matplotlib, con assi in scala uguale"*. Il collaudo è visivo: il profilo deve *sembrare* un 2412; se appare gonfio o schiacciato, cercate l'errore in `axis("equal")` o nelle formule della linea media.

Parte III

Quarto anno

La polare, le prestazioni e il diagramma di manovra

Capitolo 10

Dieci problemi risolti per il quarto anno

Nota

Dieci problemi completamente risolti sulla *meccanica del volo*: polare, volo livellato, virata, salita, planata, decollo, crociera, autonomie, inviluppo di manovra e di raffica. Il velivolo di riferimento, usato in tutte le schede, è un monomotore da addestramento: $m = 1100\,\text{kg}$, $S = 16{,}2\,\text{m}^2$, $C_{Lmax} = 1{,}6$, $C_{D0} = 0{,}025$, $k = 0{,}045$, motore da 90 kW con rendimento dell'elica $\eta = 0{,}75$; ove serve, corda media $\bar{c} = 1{,}5\,\text{m}$ e $C_{L\alpha} = 5{,}0\,\text{rad}^{-1}$. Usare *sempre* lo stesso velivolo non è pigrizia: è ciò che permette a ogni scheda di validare le altre.

10.1 Scheda 10.1 Le velocità caratteristiche

Problema. Calcolare stallo, efficienza massima e velocità di massima efficienza del velivolo di riferimento al livello del mare: i tre numeri che definiscono il carattere di un velivolo.

Formula chiave

Dal sostentamento $L = W$:

$$V_s = \sqrt{\frac{2W}{\rho S C_{Lmax}}}, \qquad V_E = \sqrt{\frac{2W}{\rho S C_L^*}}, \qquad C_L^* = \sqrt{\frac{C_{D0}}{k}}, \qquad E_{max} = \frac{1}{2\sqrt{C_{D0}k}}$$

Prompt pronto all'uso

Scrivi un programma Python di meccanica del volo. Modello: polare parabolica $C_D = C_{D0} + kC_L^2$; dal sostentamento $L = W$ ricava V_s (a C_{Lmax}), il C_L di massima efficienza $\sqrt{C_{D0}/k}$, $E_{max} = 1/(2\sqrt{C_{D0}k})$ e V_E. Dati: $m = 1100$ kg, $S = 16{,}2$ m², $\rho = 1{,}225$, $C_{Lmax} = 1{,}6$, $C_{D0} = 0{,}025$, $k = 0{,}045$. Formato: velocità in m/s e km/h. Controllo: V_s deve valere 26,07 m/s; verifica anche la relazione $V_E = V_s\sqrt{C_{Lmax}/C_L^*}$.

velocita_caratteristiche.py

```
# velocita_caratteristiche.py -- stallo, Emax e velocita' di Emax
import math

W     = 1100 * 9.80665   # peso [N]
S     = 16.2             # superficie alare [m^2]
rho   = 1.225            # livello del mare [kg/m^3]
CLmax = 1.6
CD0   = 0.025
k     = 0.045

Vs    = math.sqrt(2*W/(rho*S*CLmax))          # velocita' di stallo
CLE   = math.sqrt(CD0/k)                      # CL di massima efficienza
Emax  = 1/(2*math.sqrt(CD0*k))
VE    = math.sqrt(2*W/(rho*S*CLE))            # velocita' di Emax
```

```

print(f"Velocita' di stallo   Vs = {Vs:5.2f} m/s = {Vs*3.6:5.1f} km/h")
print(f"CL di max efficienza  CL_E = {CLE:.3f}")
print(f"Efficienza massima    Emax = {Emax:.2f}")
print(f"Velocita' di Emax     V_E = {VE:5.2f} m/s = {VE*3.6:5.1f} km/h")
```

Output

```
Velocita' di stallo   Vs = 26.07 m/s =  93.8 km/h
CL di max efficienza  CL_E = 0.745
Efficienza massima    Emax = 14.91
Velocita' di Emax     V_E = 38.19 m/s = 137.5 km/h
```

Validazione. V_s: sotto radice, $2 \cdot 10787/(1{,}225 \cdot 16{,}2 \cdot 1{,}6) = 679{,}5$, e $\sqrt{679{,}5} = 26{,}07$ m/s ✓. Si noti la proporzionalità $V_E = V_s\sqrt{C_{Lmax}/C_L^*}$: la struttura delle formule si presta a controlli incrociati che il programma rende immediati.

Prova tu

Ripetete il calcolo a 3000 m importando `atmosfera20.py` e dimostrate (numericamente e poi algebricamente) che le velocità *vere* crescono come $1/\sqrt{\sigma}$ mentre E_{max} non cambia.

10.2 Scheda 10.2 Volo livellato: la curva di potenza necessaria

Problema. Costruire la tabella $P_n(V)$ da 25 a 75 m/s e riconoscere velocità di potenza minima e "secondo regime".

Formula chiave

In volo livellato uniforme $L = W$ e $T = D$; quindi

$$C_L = \frac{2W}{\rho S V^2}, \qquad D = \tfrac{1}{2}\rho V^2 S\,(C_{D0} + kC_L^2), \qquad P_n = D\,V$$

Prompt pronto all'uso

Scrivi un programma Python con NumPy per il volo livellato. Modello: per ogni V ricava C_L dal sostentamento, C_D dalla polare parabolica, poi D e $P_n = DV$; usa vettori NumPy, non cicli. Dati: il velivolo di riferimento; V da 25 a 75 m/s a passo 5. Dominio: segnala le righe in cui $C_L > C_{Lmax}$ (punti sotto stallo, fisicamente irraggiungibili). Formato: tabella V, C_L, C_D, D, P_n in kW; in coda, minimo di P_n con argmin. Controllo: a $V = 40$ m/s deve risultare $D \simeq 727$ N.

potenza_necessaria.py

```
# potenza_necessaria.py -- curva Pn(V) in volo livellato
import numpy as np

W, S, rho = 1100*9.80665, 16.2, 1.225
CD0, k    = 0.025, 0.045

V  = np.arange(25, 76, 5, dtype=float)
CL = 2*W/(rho*S*V**2)
```

```python
CD = CD0 + k*CL**2
D  = 0.5*rho*V**2*S*CD
Pn = D*V

print("  V[m/s]     CL       CD       D[N]    Pn[kW]")
for v, cl, cd, d, p in zip(V, CL, CD, D, Pn):
    print(f"  {v:5.0f}  {cl:6.3f}  {cd:6.4f}  {d:7.1f}  {p/1000:6.2f}")

i = Pn.argmin()
print(f"\nPotenza minima {Pn[i]/1000:.2f} kW alla velocita' {V[i]:.0f} m/s")
```

Output

```
V[m/s]     CL       CD       D[N]    Pn[kW]
     25   1.739  0.1612    999.4    24.99
     30   1.208  0.0907    809.6    24.29
     35   0.887  0.0604    734.7    25.71
     40   0.679  0.0458    726.7    29.07
     45   0.537  0.0380    762.9    34.33
     50   0.435  0.0335    831.3    41.56
     55   0.359  0.0308    924.8    50.87
     60   0.302  0.0291   1039.6    62.38
     65   0.257  0.0280   1173.0    76.24
     70   0.222  0.0272   1323.2    92.62
     75   0.193  0.0267   1489.2   111.69
```

Lettura fisica. Due avvertenze dalla tabella stessa. Primo: la riga $V = 25$ m/s richiede $C_L = 1{,}74 > C_{Lmax}$: quel punto è *fisicamente irraggiungibile* (siamo sotto stallo); il programma lo calcola lo stesso, e sta a voi scartarlo — ricordate l'errore n. 3 del capitolo 7. Secondo: la resistenza è minima a 40 m/s circa (dove $E \to E_{max}$), ma la *potenza* è minima prima, a 30 m/s: sotto tale velocità, per volare più piano serve più potenza — è il secondo regime.

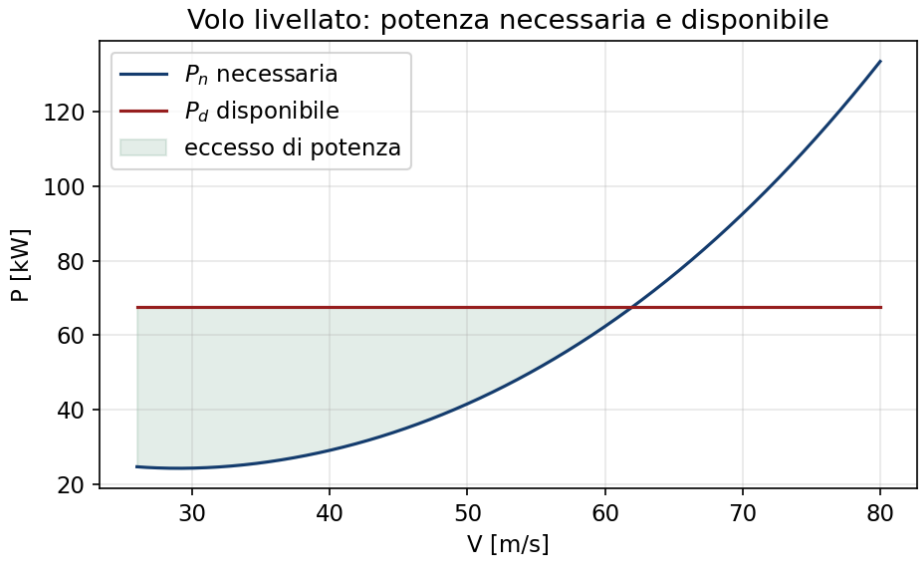


Figura 10.1. Potenza necessaria e disponibile: l'area verde è l'eccesso di potenza, motore della salita.

Prova tu

Con `np.linspace` a 500 punti, trovate con precisione la velocità di potenza minima e verificate la relazione teorica $C_L(P_{n,min}) = \sqrt{3\,C_{D0}/k}$ (attesa: $C_L = 1{,}291$).

10.3 Scheda 10.3 Prestazioni di salita

Problema. Determinare il rateo di salita al variare della velocità e individuare il rateo massimo con la relativa velocità.

Formula chiave

Dal bilancio energetico in salita quasi–stazionaria:

$$RC = \frac{P_d - P_n}{W} \qquad \text{con } P_d = \eta\, P_{motore}$$

Il rateo massimo si ha dove l'*eccesso di potenza* è massimo.

Prompt pronto all'uso

Scrivi un programma Python per le prestazioni di salita. Modello: $RC = (P_d - P_n)/W$ con $P_d = \eta P_{motore}$ costante ed elica; P_n dalla polare come nella scheda precedente. Dati: velivolo di riferimento, $P = 90$ kW, $\eta = 0{,}75$, V da 28 a 60 m/s a passo 4. Formato: tabella $V \to RC$ e riga finale con il massimo (usa argmax). Controllo: a $V = 40$ m/s, con $P_n = 29{,}07$ kW già validata, deve risultare $RC = (67500 - 29070)/10787 = 3{,}56$ m/s.

`salita.py`

```python
# salita.py -- rateo di salita al variare della velocita'
import numpy as np

W, S, rho = 1100*9.80665, 16.2, 1.225
CD0, k    = 0.025, 0.045
P_motore  = 90e3        # potenza al freno [W]
eta       = 0.75        # rendimento dell'elica

Pd = eta * P_motore                         # potenza disponibile
V  = np.arange(28, 61, 4, dtype=float)
CL = 2*W/(rho*S*V**2)
CD = CD0 + k*CL**2
Pn = 0.5*rho*V**3*S*CD
RC = (Pd - Pn)/W                            # rateo di salita [m/s]

for v, rc in zip(V, RC):
    print(f"V = {v:4.0f} m/s   RC = {rc:5.2f} m/s")
print(f"\nRateo massimo {RC.max():.2f} m/s a V = {V[RC.argmax()]:.0f} m/s")
```

Output

```
V =   28 m/s   RC =  4.01 m/s
V =   32 m/s   RC =  3.98 m/s
V =   36 m/s   RC =  3.83 m/s
V =   40 m/s   RC =  3.56 m/s
V =   44 m/s   RC =  3.19 m/s
V =   48 m/s   RC =  2.69 m/s
V =   52 m/s   RC =  2.08 m/s
V =   56 m/s   RC =  1.35 m/s
V =   60 m/s   RC =  0.47 m/s
```

Validazione (un punto, a mano). A $V = 40$ m/s: $C_L = 0{,}679$, $C_D = 0{,}0458$, $P_n = 29{,}07$ kW (dalla scheda precedente!), $P_d = 67{,}5$ kW, quindi $RC = (67500 - 29070)/10787 = 3{,}56$ m/s ✓. Riusare i risultati già validati di un'altra scheda è esso stesso un metodo di verifica.

Prova tu

Importate `atmosfera20.py` e tracciate $RC_{max}(h)$ ogni 500 m (potenza del motore aspirato $\propto \sigma$): la quota alla quale RC_{max} si annulla è la *quota di tangenza teorica* del velivolo.

10.4 Scheda 10.4 La virata corretta

Problema. Per angoli di banco di 15°, 30°, 45° e 60° calcolare fattore di carico, velocità di stallo in virata, raggio e velocità angolare della virata percorsa a $V = 45$ m/s.

Formula chiave

Nella virata corretta (orizzontale, coordinata) la portanza deve equilibrare peso e fornire la forza centripeta: $L\cos\varphi = W$ e $L\sin\varphi = \frac{mV^2}{R}$, da cui

$$n = \frac{1}{\cos\varphi}, \qquad V_{s,\varphi} = V_s\sqrt{n}, \qquad R = \frac{V^2}{g\tan\varphi}, \qquad \dot\psi = \frac{V}{R}$$

La prima è una *dimostrazione* in due righe: dividendo le due equazioni di equilibrio, $\tan\varphi = V^2/(gR)$; dalla prima, $n = L/W = 1/\cos\varphi$.

Prompt pronto all'uso

Scrivi un programma Python sulla virata corretta. Modello: equilibrio $L\cos\varphi = W$, $L\sin\varphi = mV^2/R$; niente approssimazioni. Dati: velivolo di riferimento, $\varphi = 15°, 30°, 45°, 60°$, virata a $V = 45$ m/s. Dominio: $0 < \varphi < 90°$; converti i gradi con math.radians. Formato: tabella φ, n, $V_{s,\varphi}$, R, $\dot\psi$ in gradi al secondo. Controlli: a 60° deve risultare esattamente $n = 2$ (coseno di 60° = 1/2) e a 45° $R = V^2/g$.

`virata.py`

```python
# virata.py -- fattore di carico, stallo, raggio e rateo di virata
import math

W, S, rho = 1100*9.80665, 16.2, 1.225
CLmax     = 1.6
V         = 45.0                        # velocita' di virata [m/s]
```

```
Vs        = math.sqrt(2*W/(rho*S*CLmax))

print("phi[gr]    n     Vs_phi[m/s]   R[m]    psi_p[gr/s]")
for phi in [15, 30, 45, 60]:
    fr  = math.radians(phi)
    n   = 1/math.cos(fr)              # fattore di carico
    Vsv = Vs*math.sqrt(n)             # stallo in virata
    R   = V**2/(9.80665*math.tan(fr)) # raggio di virata
    psi = math.degrees(V/R)           # velocita' angolare
    print(f"  {phi:3d}   {n:6.3f}    {Vsv:6.2f}     {R:7.1f}    {psi:6.2f}")
```

Output

```
phi[gr]    n     Vs_phi[m/s]   R[m]    psi_p[gr/s]
   15    1.035     26.52       770.6      3.35
   30    1.155     28.01       357.7      7.21
   45    1.414     31.00       206.5     12.49
   60    2.000     36.86       119.2     21.63
```

Validazione. A 60°: $n = 1/\cos 60° = 2$ *esatto*, e $V_{s,60°} = 26{,}07\sqrt{2} = 36{,}86$ m/s ✓. A 45°: $\tan 45° = 1$, dunque $R = V^2/g = 2025/9{,}80665 = 206{,}5$ m ✓. Lettura operativa: a 60° di banco lo stallo sale del 41% — è il motivo per cui le virate strette a bassa velocità, in circuito, sono la prima causa storica di incidenti.

Prova tu

A quale velocità la virata a 60° porta lo stallo a coincidere con la velocità di volo (virata al limite)? Impostate l'equazione $V = V_s\sqrt{n}$ e risolvetela; poi confrontate con la velocità di manovra V_A della scheda 10.9: la coincidenza non è casuale.

10.5 Scheda 10.5 Volo librato: quanto lontano si arriva senza motore

Problema. In caso di piantata motore a 2000 m, calcolare angolo di planata minimo e distanza massima percorribile in aria calma.

Formula chiave

In planata stazionaria $\tan\gamma = 1/E$: l'angolo di discesa minimo si ha a E_{max} e la distanza percorribile da quota h è $d = h\,E_{max}$ (in aria calma).

Prompt pronto all'uso

Scrivi un programma Python sul volo librato. Modello: equilibrio in planata stazionaria, $\tan\gamma = 1/E$*, distanza* $d = hE$*; usa* E_{max} *dalla polare. Dati:* $C_{D0} = 0{,}025$, $k = 0{,}045$, *quota iniziale 2000 m. Formato:* E_{max}, γ *in gradi con math.degrees, distanza in km. Controllo: la distanza deve valere* $2000 \cdot 14{,}91 \simeq 29{,}8$ *km.*

planata.py

```
# planata.py -- planata di massima distanza
import math
CD0, k = 0.025, 0.045
h = 2000.0                                  # quota iniziale [m]

```

```
Emax  = 1/(2*math.sqrt(CD0*k))
gamma = math.degrees(math.atan(1/Emax))      # angolo di rampa minimo
dist  = h * Emax                             # distanza percorribile

print(f"Efficienza massima     Emax  = {Emax:.2f}")
print(f"Angolo di planata min  gamma = {gamma:.2f} gradi")
print(f"Distanza da h = {h:.0f} m  : {dist/1000:.1f} km")
```

Output

```
Efficienza massima     Emax  = 14.91
Angolo di planata min  gamma = 3.84 gradi
Distanza da h = 2000 m  : 29.8 km
```

Validazione. $d = 2000 \cdot 14{,}91 = 29\,820$ m $\simeq 29{,}8$ km; e $\arctan(1/14{,}91) = 3{,}84°$ ✓. Trenta chilometri da duemila metri: il numero che ogni pilota di monomotore tiene a mente.

Prova tu

Aggiungete il vento: con vento frontale V_w la distanza al suolo diventa $d = h\,E\,(1 - V_w/V)$ in prima approssimazione. Tabulate d per V_w da 0 a 15 m/s planando a $V = V_E$ e discutete perché, con vento contrario, conviene planare *più veloci* di V_E.

10.6 Scheda 10.6 La corsa di decollo per integrazione numerica

Problema. Stimare la corsa di decollo su pista in asfalto: spinta statica dell'elica $T = 2600$ N (costante, in prima approssimazione), coefficiente d'attrito di rotolamento $\mu_r = 0{,}03$, assetto di rullaggio con $C_L = 0{,}4$ e $\Delta C_{D0} = 0{,}02$ per carrello e flap; rotazione a $V_R = 1{,}15\,V_s$.

Formula chiave

Secondo principio della dinamica lungo la pista:

$$m\frac{\mathrm{d}V}{\mathrm{d}t} = T - D - \mu_r\,(W - L)$$

Non esiste soluzione chiusa comoda (D e L dipendono da V^2): si integra a piccoli passi, $V \mathrel{+}= a\,\Delta t$, $s \mathrel{+}= V\,\Delta t$ — il metodo di Eulero esplicito, il più antico e istruttivo degli integratori.

Prompt pronto all'uso

Scrivi un programma Python che integri numericamente la corsa di decollo. Modello: $ma = T - D - \mu_r(W - L)$ con L e D aerodinamici a C_L di rullaggio costante; metodo di Eulero con passo $\Delta t = 0{,}01$ s; ciclo while fino a $V_R = 1{,}15\,V_s$. Dati: velivolo di riferimento, $T = 2600$ N, $\mu_r = 0{,}03$, $C_L = 0{,}4$, $\Delta C_{D0} = 0{,}02$. Dominio: proteggi il termine d'attrito con $\max(W - L, 0)$, spiega perché in un commento. Formato: V_R, corsa in metri, tempo in secondi. Controllo d'ordine di grandezza: con a media di circa 1,7 m/s², $s \approx V_R^2/(2a) \approx 260$ m; il risultato deve essere in quell'intorno.

decollo.py

```
# decollo.py -- corsa di decollo con il metodo di Eulero
import math
```

```
3
4  m, S, rho = 1100.0, 16.2, 1.225
5  W         = m*9.80665
6  CD0, k    = 0.025, 0.045
7  T_elica   = 2600.0      # spinta statica [N] (costante: ipotesi forte)
8  mu_r      = 0.03        # attrito di rotolamento (asfalto)
9  CL_g      = 0.4         # assetto di rullaggio
10 dCD0      = 0.02        # carrello + flap estratti
11
12 Vs = math.sqrt(2*W/(rho*S*1.6))
13 VR = 1.15*Vs                          # velocita' di rotazione
14 dt = 0.01
15 V, s, t = 0.0, 0.0, 0.0
16 while V < VR:
17     L = 0.5*rho*V**2*S*CL_g
18     D = 0.5*rho*V**2*S*(CD0 + dCD0 + k*CL_g**2)
19     a = (T_elica - D - mu_r*max(W - L, 0.0))/m
20     V += a*dt                          # Eulero esplicito
21     s += V*dt
22     t += dt
23
24 print(f"Vs = {Vs:.2f} m/s   VR = {VR:.2f} m/s")
25 print(f"Corsa di decollo s = {s:.0f} m   in t = {t:.1f} s")
```

Output

```
Vs = 26.07 m/s   VR = 29.98 m/s
Corsa di decollo s = 236 m   in t = 15.3 s
```

Validazione. Stima chiusa ad accelerazione media: in partenza $a_0 = (2600 - 0{,}03 \cdot 10787)/1100 = 2{,}07\ \text{m/s}^2$, a V_R l'aerodinamica la riduce; con $a_m \simeq 1{,}9\ \text{m/s}^2$, $s \simeq V_R^2/(2a_m) = 899/3{,}8 \simeq 237$ m: l'integrazione numerica (236 m) sta esattamente dove la stima la colloca ✓. Il metodo di Eulero è approssimato, ma con $\Delta t = 0{,}01$ s su una manovra di 15 secondi l'errore di integrazione è ben minore dell'incertezza sull'ipotesi di spinta costante: giudicare *quale* errore domina è cultura ingegneristica.

Prova tu

Rendete la spinta realistica: $T(V) = T_0\,(1 - V/120)$ (l'elica "perde" spinta con la velocità). Ripetete il calcolo, poi dimezzate il passo a $\Delta t = 0{,}005$: se il risultato cambia di poco, la soluzione è *convergente* — il collaudo specifico dei metodi numerici.

10.7 Scheda 10.7 Crociera: autonomia oraria o chilometrica?

Problema. Confrontare i due regimi di crociera del velivolo a elica: volo alla velocità di potenza minima (massima *durata*) e volo a E_{max} (massima *distanza*), calcolando per ciascuno il consumo orario e per cento chilometri. Consumo specifico $c_s = 0{,}30$ kg/kWh riferito alla potenza al freno.

Formula chiave

La portata di combustibile è $\dot{m}_f = c_s\, P_{motore} = c_s\, P_n/\eta$. Quindi: consumo orario minimo dove P_n è minima (cioè a $C_L = \sqrt{3C_{D0}/k}$); consumo *chilometrico* $\dot{m}_f/V = c_s D/\eta$ minimo dove D è minima, cioè a E_{max}. Due ottimi diversi per due domande diverse.

Prompt pronto all'uso

Scrivi un programma Python che confronti i due regimi di crociera di un velivolo a elica. Modello: portata combustibile $\dot{m}_f = c_s P_n/\eta$ con c_s in kg/(W·s); regime di massima durata a $C_L = \sqrt{3C_{D0}/k}$, regime di massima distanza a $C_L = \sqrt{C_{D0}/k}$. Dati: velivolo di riferimento, c_s = 0,30 kg/kWh (convertila tu in unità SI, mostrando la conversione in un commento). Formato: per ciascun regime V, P_n, kg/h e kg/100 km. Controllo: il regime di massima durata deve consumare meno all'ora ma più al chilometro; se non accade, c'è un errore.

`crociera.py`

```
# crociera.py -- massima autonomia oraria vs chilometrica
import math

W, S, rho = 1100*9.80665, 16.2, 1.225
CD0, k    = 0.025, 0.045
eta       = 0.75
cs        = 0.30/3.6e6    # 0.30 kg/kWh -> kg/(W s): /1000 e /3600

def regime(nome, CL):
    V  = math.sqrt(2*W/(rho*S*CL))
    D  = 0.5*rho*V**2*S*(CD0 + k*CL**2)
    Pn = D*V
    kg_h  = cs*Pn/eta*3600          # consumo orario
    kg_km = cs*Pn/eta*1000/V        # consumo al chilometro
    print(f"{nome:28s} V={V:5.2f} m/s  Pn={Pn/1000:6.2f} kW  "
          f"{kg_h:5.2f} kg/h  {kg_km*100:5.2f} kg/100km")

regime("Max durata (Pn minima)",  math.sqrt(3*CD0/k))
regime("Max distanza (E max)",    math.sqrt(CD0/k))
```

Output

```
Max durata (Pn minima)       V=29.02 m/s  Pn= 24.25 kW   9.70 kg/h   9.28 kg/100
    km
Max distanza (E max)         V=38.19 m/s  Pn= 27.64 kW  11.05 kg/h   8.04 kg/100
    km
```

Validazione. Il controllo qualitativo del prompt è soddisfatto: la massima durata consuma meno all'ora (9,70 < 11,05) ma più al chilometro (9,28 > 8,04). Quantitativamente: a E_{max}, $D = W/E_{max} = 10787/14{,}91 = 723{,}5$ N e il consumo chilometrico $c_s D/\eta = 8{,}33 \cdot 10^{-8} \cdot 723{,}5/0{,}75 = 8{,}04 \times 10^{-5}$ kg/m = 8,04 kg/100km ✓. La sorveglianza antincendio vola al primo regime; il trasferimento al secondo.

Prova tu

Tabulate kg/h e kg/100 km per V da 28 a 60 m/s e tracciate le due curve: i minimi devono cadere alle due velocità della tabella. Osservate quanto è *piatta* la curva chilometrica intorno al minimo: volare il 10% più veloci di V_E costa pochissimo, ed è ciò che si fa in pratica.

10.8 Scheda 10.8 Autonomia di Breguet

Problema. Con 120 kg di combustibile a bordo (massa iniziale 1100 kg), calcolare distanza e durata massime tenendo conto che il velivolo *si alleggerisce* mentre consuma.

Formula chiave

Per il velivolo a elica, integrando $\mathrm{d}W = -g\,c_s P_n/\eta\ \mathrm{d}t$ lungo la crociera a C_L costante si ottengono le formule di Breguet:

$$d = \frac{\eta}{g\,c_s}\,E\,\ln\frac{W_0}{W_1}, \qquad t = \frac{\eta}{g\,c_s}\,\frac{C_L^{3/2}}{C_D}\,\sqrt{2\rho S}\left(\frac{1}{\sqrt{W_1}} - \frac{1}{\sqrt{W_0}}\right)$$

con E massimo per la distanza e $C_L^{3/2}/C_D$ massimo per la durata. La comparsa del logaritmo è la firma dell'alleggerimento progressivo.

Prompt pronto all'uso

Scrivi un programma Python con le formule di Breguet per velivolo a elica. Modello: le due formule con c_s in kg/(W·s) e i pesi in newton; per la distanza usa E_{max}, per la durata il massimo di $C_L^{3/2}/C_D$ (a $C_L = \sqrt{3C_{D0}/k}$). Dati: velivolo di riferimento, $m_0 = 1100$ kg, combustibile 120 kg, $c_s = 0{,}30$ kg/kWh, $\eta = 0{,}75$. Formato: distanza in km, durata in ore, entrambe con confronto rispetto alla stima "ingenua" a peso costante. Controllo: la stima ingenua della distanza è 120 kg diviso 8,04 kg/100 km $\simeq$ 1493 km; Breguet deve dare di più, perché il velivolo si alleggerisce.

autonomia.py

```python
# autonomia.py -- distanza e durata massime (Breguet, elica)
import math

S, rho   = 16.2, 1.225
CD0, k   = 0.025, 0.045
eta      = 0.75
cs       = 0.30/3.6e6          # kg/(W s)
g        = 9.80665
m0, mf   = 1100.0, 120.0
W0, W1   = m0*g, (m0 - mf)*g

Emax = 1/(2*math.sqrt(CD0*k))
dist = eta/(g*cs) * Emax * math.log(W0/W1)

CLd  = math.sqrt(3*CD0/k)                  # CL di massima durata
F    = CLd**1.5/(CD0 + k*CLd**2)           # CL^(3/2)/CD massimo
dur  = eta/(g*cs) * F * math.sqrt(2*rho*S) * (W1**-0.5 - W0**-0.5)

print(f"Emax = {Emax:.2f}   CL^1.5/CD max = {F:.3f}")
print(f"Distanza massima (Breguet) : {dist/1000:6.0f} km")
print(f"Durata massima  (Breguet)  : {dur/3600:6.2f} h")
```

Output

```
Emax = 14.91   CL^1.5/CD max = 14.669
Distanza massima (Breguet) :   1580 km
Durata massima  (Breguet)  :  13.49 h
```

Validazione. Stima ingenua a peso costante: $120/8{,}04 \cdot 100 = 1493$ km e $120/9{,}70 = 12{,}4$ h (consumi della scheda 10.7). Breguet dà il 6–9% in più in entrambi i casi, *come deve*: consumando, il velivolo pesa meno, richiede meno potenza e consuma meno. La stima ingenua fa da minorante rigoroso — se Breguet desse *meno*, il programma sarebbe sbagliato. Ecco un collaudo che non richiede alcun valore tabellato: solo fisica.

Prova tu

Ricavate dalla formula della distanza il "raggio d'azione" (andata e ritorno con riserva del 20% di combustibile) e calcolate quanta massa di combustibile servirebbe per raddoppiare la distanza: il logaritmo vi mostrerà una crescita più che lineare — la tirannia dell'equazione di Breguet, la stessa che governa i razzi.

10.9 Scheda 10.9 Il diagramma di manovra

Problema. Calcolare i punti notevoli dell'inviluppo di volo secondo CS-23, categoria normale ($n_{max} = 3{,}8$, $n_{min} = -1{,}52$, $V_D = 85$ m/s).

Formula chiave

Il ramo di stallo positivo è la parabola $n(V) = \frac{1}{2}\rho V^2 S\, C_{Lmax}/W$; la *velocità di manovra* è il suo incontro con n_{max}:

$$V_A = V_s\sqrt{n_{max}}$$

Sotto V_A il velivolo stalla prima di raggiungere n_{max}: la struttura è protetta aerodinamicamente.

Prompt pronto all'uso

Scrivi un programma Python per il diagramma di manovra CS-23. Modello: ramo di stallo $n = \frac{1}{2}\rho V^2 S C_{Lmax}/W$, troncato a n_{max}; $V_A = V_s\sqrt{n_{max}}$. Dati: velivolo di riferimento, $n_{max} = 3{,}8$, $n_{min} = -1{,}52$, $V_D = 85$ m/s. Formato: V_s, V_A, V_D in m/s e km/h, poi tabella $V \to n$ di stallo per $V = 30, 40, 50, 60$ con min() per il troncamento. Controllo: la parabola deve passare per $(V_s, 1)$ e per (V_A, n_{max}) — verificalo.

diagramma_manovra.py

```python
# diagramma_manovra.py -- punti notevoli del diagramma di manovra
import math

W, S, rho = 1100*9.80665, 16.2, 1.225
CLmax     = 1.6
n_max     = 3.8          # categoria normale (CS-23)
n_min     = -1.52
VD        = 85.0         # velocita' di affondata [m/s]

Vs = math.sqrt(2*W/(rho*S*CLmax))       # stallo a n = 1
VA = Vs*math.sqrt(n_max)                # velocita' di manovra

print(f"Vs = {Vs:5.2f} m/s = {Vs*3.6:6.1f} km/h")
print(f"VA = {VA:5.2f} m/s = {VA*3.6:6.1f} km/h")
print(f"VD = {VD:5.2f} m/s = {VD*3.6:6.1f} km/h")
print("\n  V[m/s]   n stallo")
for V in [30, 40, 50, 60]:
    n = 0.5*rho*V**2*S*CLmax/W          # fattore di carico al CLmax
    print(f"  {V:5.0f}    {min(n, n_max):5.2f}")
```

```
Output
```

```
Vs = 26.07 m/s =   93.8 km/h
VA = 50.81 m/s =  182.9 km/h
VD = 85.00 m/s =  306.0 km/h

  V[m/s]   n stallo
      30      1.32
      40      2.35
      50      3.68
      60      3.80
```

Validazione. $V_A = 26{,}07\sqrt{3{,}8} = 26{,}07 \cdot 1{,}949 = 50{,}81$ m/s ✓. E infatti a 50 m/s il fattore di carico allo stallo vale $3{,}68 < 3{,}8$, a 60 m/s supererebbe il limite ed è troncato a n_{max}: la tabella riproduce la geometria del diagramma.

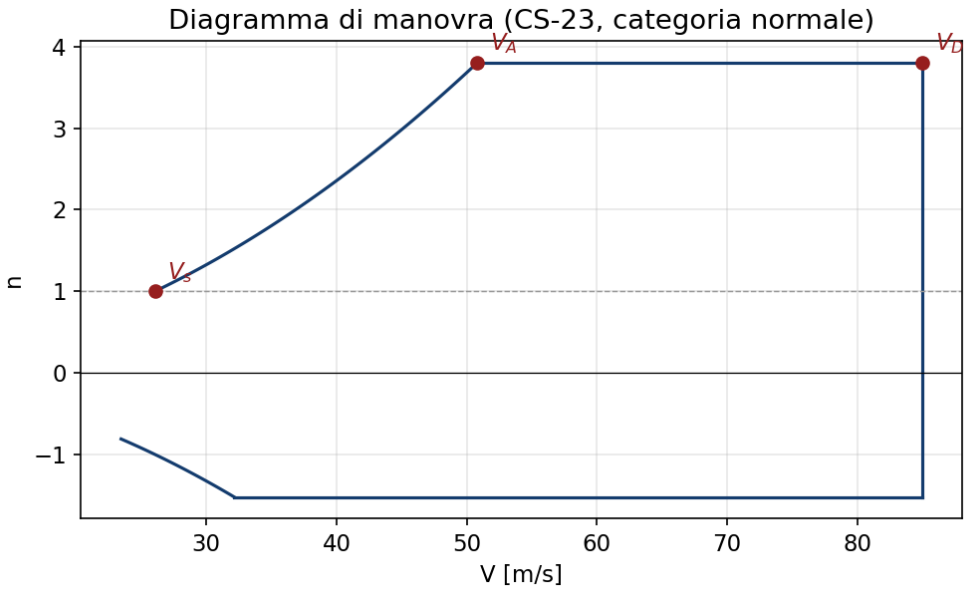


Figura 10.2. Il diagramma di manovra tracciato dai valori calcolati.

Dialoga con l'IA

Fate tracciare all'IA il diagramma completo (rami di stallo positivo e negativo, limiti n_{max}, n_{min}, chiusura a V_D) e collaudatelo su tre punti: la parabola positiva deve passare per $(V_s, 1)$ e per (V_A, n_{max}), e il vertice destro deve stare in (V_D, n_{max}). Tre punti giusti su una parabola nota: collaudo sufficiente.

10.10 Scheda 10.10 Il fattore di carico da raffica

Problema. Calcolare il fattore di carico dovuto a una raffica verticale secondo lo schema CS-23: raffica $U_{de} = 15{,}24$ m/s (50 ft/s) alla velocità di crociera $V_C = 55$ m/s e $U_{de} = 7{,}62$ m/s (25 ft/s) a $V_D = 85$ m/s.

Formula chiave

La raffica verticale U varia l'incidenza di $\Delta\alpha \simeq U/V$ e quindi il C_L di $C_{L\alpha}\,U/V$; il fattore di carico diventa

$$n = 1 \pm \frac{\rho_0\, U_{de}\, V\, C_{L\alpha}\, K_g}{2\,(W/S)}, \qquad K_g = \frac{0{,}88\,\mu_g}{5{,}3 + \mu_g}, \quad \mu_g = \frac{2\,(W/S)}{\rho\,\bar{c}\,g\,C_{L\alpha}}$$

$K_g < 1$ è il fattore di attenuazione: il velivolo, entrando gradualmente nella raffica, non ne sente tutto il valore nominale. Si noti il ruolo *protettivo* del carico alare W/S a denominatore.

Prompt pronto all'uso

Scrivi un programma Python per il fattore di carico da raffica secondo CS-23. Modello: formula di Pratt con attenuazione $K_g = 0{,}88\mu_g/(5{,}3 + \mu_g)$ e $\mu_g = 2(W/S)/(\rho\bar{c}gC_{L\alpha})$. Dati: velivolo di riferimento, $\bar{c} = 1{,}5$ m, $C_{L\alpha} = 5{,}0$ rad^{-1}; casi ($V_C = 55$ m/s, $U_{de} = 15{,}24$ m/s) e ($V_D = 85$ m/s, $U_{de} = 7{,}62$ m/s). Formato: per ciascun caso μ_g, K_g, Δn e i due fattori $n^{\pm} = 1 \pm \Delta n$. Controllo: Δn deve crescere linearmente con V e con U_{de}: verifica che il caso V_C dia Δn maggiore del caso V_D nonostante la velocità minore.

raffica.py

```
# raffica.py -- fattore di carico da raffica (CS-23, formula di Pratt)
W, S, rho = 1100*9.80665, 16.2, 1.225
CLa   = 5.0            # pendenza della retta di portanza [1/rad]
mac   = 1.5            # corda media aerodinamica [m]
g     = 9.80665

ws  = W/S                                   # carico alare [N/m^2]
mug = 2*ws/(rho*mac*g*CLa)                  # parametro di massa
Kg  = 0.88*mug/(5.3 + mug)                  # attenuazione

print(f"Carico alare W/S = {ws:.1f} N/m^2   mu_g = {mug:.2f}"
      f"   Kg = {Kg:.4f}\n")
for V, Ude in [(55.0, 15.24), (85.0, 7.62)]:
    dn = rho*Ude*V*CLa*Kg/(2*ws)
    print(f"V = {V:5.1f} m/s  Ude = {Ude:5.2f} m/s  "
          f"dn = {dn:.3f}  ->  n+ = {1+dn:.2f}   n- = {1-dn:.2f}")
```

Output

```
Carico alare W/S = 665.9 N/m^2   mu_g = 14.78   Kg = 0.6477

V =  55.0 m/s  Ude = 15.24 m/s  dn = 2.497  ->  n+ = 3.50   n- = -1.50
V =  85.0 m/s  Ude =  7.62 m/s  dn = 1.930  ->  n+ = 2.93   n- = -0.93
```

Validazione. La linearità richiesta dal prompt è verificabile a mente: passando da V_C a V_D la velocità cresce del fattore $85/55 = 1{,}545$ mentre la raffica si dimezza, quindi $\Delta n_{V_D}/\Delta n_{V_C} = 1{,}545/2 = 0{,}773$; e infatti $1{,}930/2{,}497 = 0{,}773$ ✓. Confronto strutturale con la scheda precedente: a V_C la raffica produce $n^+ = 3{,}50 < 3{,}8$ — per questo velivolo dimensiona la *manovra*, non la raffica. Su un velivolo con basso carico alare accadrebbe l'opposto: rifate il conto con W/S dimezzato per convincervene.

Prova tu

Sovrapponete le *linee di raffica* (rette per $n = 1$ a $V = 0$) al diagramma di manovra della scheda precedente con Matplotlib. L'inviluppo combinato manovra+raffica è esattamente il disegno richiesto dalle prove ministeriali sull'argomento.

Parte IV

Quinto anno

Le strutture, il flusso comprimibile e la prova d'esame

Capitolo II

Dieci problemi risolti per il quinto anno

Nota

Dieci problemi completamente risolti su strutture e gasdinamica: longherone, geometria delle sezioni, flussi di taglio, diagrammi lungo l'ala, torsione del cassone, giunzioni rivettate, aste di controventatura, fusoliera pressurizzata, moto isoentropico e onde d'urto — e, insieme, la preparazione alla seconda prova. Il quinto anno è l'anno del rigore: qui ogni scheda è una piccola relazione di calcolo, e ogni prompt è un documento di specifica.

II.1 Scheda II.1 Il longherone a I idealizzato

Problema. Un longherone a I idealizzato ha solette 40×5 mm a interasse $d = 120$ mm; nella sezione agisce $M = 6{,}0$ kN m. Calcolare la tensione nelle solette (formula di Navier, anima reagente al solo taglio) e verificare rispetto alla lega 2024-T3.

Formula chiave

Per la sezione idealizzata a due solette di area A a interasse d (anima reagente solo a taglio):

$$I = 2A\left(\frac{d}{2}\right)^2 = \frac{A\,d^2}{2}, \qquad \sigma = \frac{M\,(d/2)}{I} = \frac{M}{A\,d}$$

La seconda forma ha una lettura fisica immediata: il momento si scompone nella coppia di forze $F = M/d$ nelle due solette, e $\sigma = F/A$.

Prompt pronto all'uso

Scrivi un programma Python di scienza delle costruzioni. Modello: longherone idealizzato a due solette concentrate, anima non reagente a flessione; Navier $\sigma = Mc/I$ con $I = 2A(d/2)^2$. Dati: solette 40×5 mm, interasse 120 mm, $M = 6{,}0$ kN·m; lavora in unità SI e converti in mm e MPa solo in stampa. Formato: area in mm², inerzia in mm⁴, tensione in MPa. Controllo doppio: la tensione deve coincidere con la via della coppia di forze, $\sigma = M/(Ad) = 250$ MPa.

longherone.py

```
# longherone.py -- tensione di flessione in un longherone a I idealizzato
b, t = 0.040, 0.005      # soletta: larghezza e spessore [m]
d    = 0.120             # interasse fra le solette [m]
M    = 6.0e3             # momento flettente nella sezione [N m]

A = b * t                            # area di una soletta [m^2]
I = 2 * A * (d/2)**2                 # inerzia (anima trascurata) [m^4]
sigma = M * (d/2) / I                # formula di Navier [Pa]

print(f"Area soletta   A = {A*1e6:6.0f} mm^2")
```

```
print(f"Inerzia        I = {I*1e12:10.0f} mm^4")
print(f"Tensione   sigma = {sigma/1e6:6.1f} MPa")
```

Output

```
Area soletta   A =    200 mm^2
Inerzia        I =    1440000 mm^4
Tensione   sigma =  250.0 MPa
```

Validazione (per la via fisica). $F = M/d = 6000/0{,}12 = 50\,000$ N; $\sigma = F/A = 50000/(200 \cdot 10^{-6}) = 250$ MPa ✓. Per una lega 2024-T3 ($\sigma_{sn} \simeq 345$ MPa) il margine di sicurezza rispetto allo snervamento vale $MS = 345/250 - 1 = 0{,}38$: positivo, la sezione è verificata.

Prova tu

Trasformate il calcolo in funzione `def sigma_longherone(M, A, d):` e tabulate σ per M da 0 a 10 kN m: a quale momento si raggiunge lo snervamento?

11.2 Scheda 11.2 Geometria delle sezioni composte

Problema. Una sezione a T è composta da un'ala orizzontale 80×10 mm sopra un'anima 10×60 mm (quote dei baricentri parziali: 65 e 30 mm dalla base). Calcolare baricentro e momento d'inerzia baricentrico con il teorema di trasposizione, in un programma valido per qualunque sezione a rettangoli.

Formula chiave

$$y_G = \frac{\sum A_i\, y_i}{\sum A_i}, \qquad I = \sum \left[\frac{b_i h_i^3}{12} + A_i\,(y_i - y_G)^2 \right] \quad \text{(Huygens–Steiner)}$$

Prompt pronto all'uso

Scrivi un programma Python con NumPy per la geometria delle sezioni. Modello: sezione composta da rettangoli descritti da tre vettori (basi, altezze, quote dei baricentri parziali); baricentro come media pesata sulle aree, inerzia con Huygens–Steiner. Dati: ala 80×10 a quota 65 mm, anima 10×60 a quota 30 mm. Formato: area totale, y_G, I in mm⁴. Il programma deve funzionare per un numero qualsiasi di rettangoli senza modifiche. Controllo: il caso del rettangolo unico $b \times h$ deve restituire $y_G = h/2$ e $I = bh^3/12$.

`sezione_composta.py`

```
# sezione_composta.py -- baricentro e inerzia di una sezione a T
# rettangolo 1: ala orizzontale 80x10 mm; rettangolo 2: anima 10x60 mm
import numpy as np

b  = np.array([80.0, 10.0])        # basi [mm]
h  = np.array([10.0, 60.0])        # altezze [mm]
yc = np.array([65.0, 30.0])        # quote dei baricentri parziali [mm]

A  = b * h                                  # aree parziali
yG = (A * yc).sum() / A.sum()               # baricentro della sezione
Ig = b * h**3 / 12                          # inerzie proprie
I  = (Ig + A * (yc - yG)**2).sum()          # teorema di Huygens
```

```

print(f"Area totale  A  = {A.sum():7.0f} mm^2")
print(f"Baricentro   yG = {yG:7.2f} mm")
print(f"Inerzia      I  = {I:10.0f} mm^4")
```

Output

```
Area totale  A  =    1400 mm^2
Baricentro   yG =   50.00 mm
Inerzia      I  =     606667 mm^4
```

Validazione. $y_G = (800\cdot 65 + 600\cdot 30)/1400 = 70000/1400 = 50$ mm ✓; e $I = 6667 + 800\cdot 15^2 + 180000 + 600\cdot 20^2 = 6667 + 180000 + 180000 + 240000 = 606\,667$ mm^4 ✓. Il programma vale per *qualunque* sezione a rettangoli: basta allungare i tre vettori. È il vostro primo strumento generale di ufficio tecnico.

Prova tu

Eseguite il collaudo richiesto dal prompt (rettangolo unico) e poi calcolate la sezione a doppio T 100×120 con spessori 8 mm: per simmetria deve risultare $y_G = 60$ mm — un controllo che non richiede alcun calcolo.

11.3 Scheda 11.3 Il flusso di taglio nell'anima del longherone

Problema. Nel longherone della scheda 11.1 la sezione trasmette un taglio $T = 8$ kN. Nella schematizzazione a solette concentrate, calcolare il flusso di taglio nell'anima (spessore $t = 1{,}5$ mm) e la tensione tangenziale.

Formula chiave

Con le aree di flessione concentrate nelle solette, il flusso di taglio è *costante* lungo l'anima e vale

$$q = \frac{T}{d}, \qquad \tau = \frac{q}{t}$$

Dimostrazione per equilibrio: il momento delle forze elementari $q\,\mathrm{d}s$ rispetto a una soletta deve equilibrare T posto all'interasse; essendo q costante, $q\,d = T$. È il caso limite della formula di Jourawski quando l'anima non contribuisce a I.

Prompt pronto all'uso

Scrivi un programma Python sul flusso di taglio. Modello: longherone idealizzato a due solette, flusso costante $q = T/d$ nell'anima, tensione $\tau = q/t$. Dati: $T = 8$ kN, $d = 120$ mm, $t = 1{,}5$ mm. Formato: q in N/mm e τ in MPa, più il confronto con la τ ammissibile di 100 MPa e il margine di sicurezza. Controllo dimensionale: q deve valere $8000/120 = 66{,}67$ N/mm — verifica che stampando q in kN/m ottieni lo stesso numero (N/mm e kN/m coincidono).

flusso_taglio.py

```
# flusso_taglio.py -- flusso di taglio nell'anima (solette concentrate)
T = 8.0e3        # taglio nella sezione [N]
d = 0.120        # interasse solette [m]
t = 1.5e-3       # spessore dell'anima [m]
tau_amm = 100e6  # ammissibile a taglio dell'anima [Pa]

```

```
q   = T / d                    # flusso di taglio [N/m], costante
tau = q / t                    # tensione tangenziale [Pa]

print(f"Flusso di taglio q  = {q/1000:6.2f} N/mm  (= {q/1000:.2f} kN/m)")
print(f"Tensione         tau = {tau/1e6:6.1f} MPa")
print(f"Margine di sicurezza = {tau_amm/tau - 1:6.2f}")
```

Output

```
Flusso di taglio q  =  66.67 N/mm  (=  66.67 kN/m)
Tensione         tau =   44.4 MPa
Margine di sicurezza =   1.25
```

Validazione. $q = 8000/120 = 66{,}67$ N/mm e $\tau = 66{,}67/1{,}5 = 44{,}4$ MPa ✓. Notare la coincidenza numerica N/mm ≡ kN/m richiesta dal prompt: le uguaglianze fra unità sono collaudi *gratuiti*, e il flusso q — forza per unità di lunghezza — è la grandezza regina delle strutture a guscio che incontrerete in ogni testo di costruzioni.

Prova tu

Componete le due verifiche del longherone: leggete $T(y)$ e $M(y)$ dalla scheda 11.4 e, sezione per sezione, calcolate σ nelle solette e τ nell'anima. Dove è più critica la flessione e dove il taglio?

11.4 Scheda 11.4 Taglio e momento lungo la semiala

Problema. Integrare numericamente una distribuzione di portanza ellittica ($L_{tot} = n\,W$ con $n = 2$, apertura $b = 11$ m) per ottenere $T(y)$ e $M(y)$ lungo la semiala: il cuore della prova d'esame.

Formula chiave

Dalla teoria della trave, con carico distribuito $\ell(y)$ (qui la portanza per unità di apertura) e ala scarica all'estremità:

$$T(y) = \int_y^s \ell(\xi)\,\mathrm{d}\xi, \qquad M(y) = \int_y^s \ell(\xi)\,(\xi - y)\,\mathrm{d}\xi$$

Per distribuzione ellittica, $\ell(y) = \ell_0\sqrt{1-(y/s)^2}$ con $\ell_0 = \dfrac{4\,(L/2)}{\pi s}$.

Prompt pronto all'uso

Scrivi un programma Python con NumPy per i diagrammi di sollecitazione dell'ala. Modello: distribuzione ellittica di portanza normalizzata a L/2 sulla semiala; taglio e momento come integrali dal tip alla sezione corrente, calcolati con np.trapezoid su 501 punti. Dati: $L_{tot} = 2 \times 10787$ N ($n = 2$), $b = 11$ m; sezioni di stampa ogni 1,1 m. Formato: tabella y, T, M. Controlli analitici: $T(0)$ deve valere esattamente $L/2 = 10787$ N e $M(0) = T(0) \cdot 4s/(3\pi)$, il braccio del baricentro della semi-ellisse.

taglio_flessione_ala.py

```
# taglio_flessione_ala.py -- T(y) e M(y) per ala con carico ellittico
import numpy as np

L_tot = 2 * 10787.0      # portanza totale = n*W con n=2 [N]
```

```
b     = 11.0            # apertura alare [m]
s     = b/2             # semiapertura

y  = np.linspace(0, s, 6)
# distribuzione ellittica: l(y) = l0*sqrt(1-(y/s)^2), integrale = L/2
l0 = L_tot/2 * 4/(np.pi*s)

# integrazione numerica dal tip verso la radice
yy = np.linspace(0, s, 501)
ll = l0*np.sqrt(1-(yy/s)**2)
print("  y[m]     T[N]      M[Nm]")
for yi in y:
    m  = yy >= yi
    T  = np.trapezoid(ll[m], yy[m])
    Mf = np.trapezoid(ll[m]*(yy[m]-yi), yy[m])
    print(f"  {yi:4.1f}  {T:8.0f}  {Mf:9.0f}")
```

Output

```
y[m]      T[N]       M[Nm]
   0.0     10787       25178
   1.1      8031       14818
   2.2      5418        7408
   3.3      3049        2755
   4.4      1106         498
   5.5         0           0
```

Validazione. Due controlli analitici esatti. In radice il taglio deve valere l'intera portanza della semiala: $T(0) = L/2 = 10\,787$ N ✓. E il momento in radice è $T(0)$ per la distanza del baricentro della semi–ellisse, $\bar{y} = 4s/(3\pi) = 2{,}334$ m: $M(0) = 10787 \cdot 2{,}334 = 25\,177$ N m ✓ (lo scarto di 1 su 25178 è l'errore di integrazione trapezoidale con 501 punti).

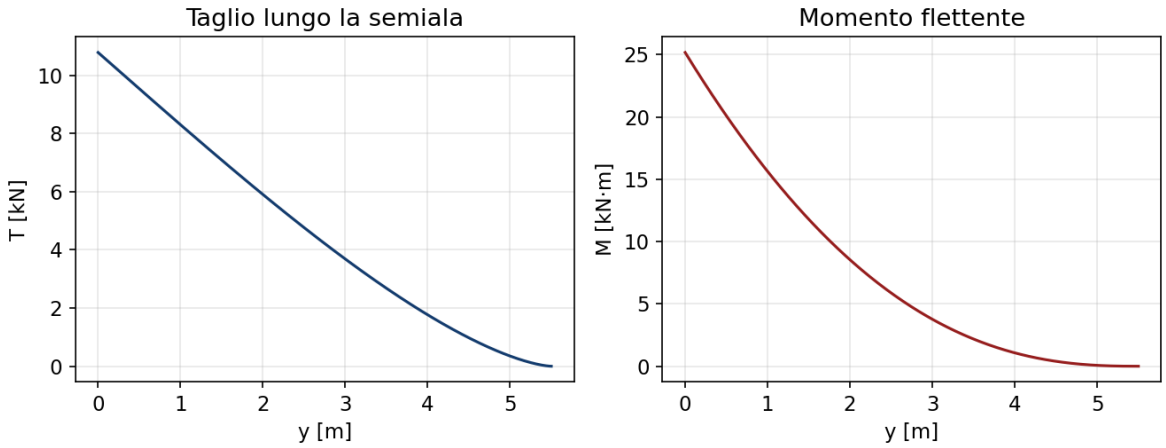


Figura 11.1. Diagrammi di taglio e momento lungo la semiala: entrambe le curve si annullano all'estremità e crescono verso la radice, dove la struttura va dimensionata.

Prova tu

Aggiungete il *momento di sgravio*: un serbatoio di 60 kg a $y = 2$ m e la massa strutturale dell'ala distribuita uniformemente (90 kg per semiala), entrambe moltiplicate per $n = 2$. Verificate che il momento in radice diminuisca: è il motivo per cui il carburante si imbarca nelle ali.

11.5 Scheda 11.5 Torsione del cassone alare: formula di Bredt

Problema. Il cassone monocella di un'ala, schematizzato come rettangolo di 600×200 mm con pareti di spessore uniforme $t = 1{,}2$ mm, trasmette un momento torcente $M_t = 4{,}0$ kN m. Calcolare tensione tangenziale e angolo unitario di torsione ($G = 27$ GPa).

Formula chiave

Per la sezione chiusa in parete sottile (prima e seconda formula di Bredt):

$$\tau = \frac{M_t}{2\,\Omega\,t}, \qquad \theta = \frac{M_t}{4\,\Omega^2 G}\oint \frac{\mathrm{d}s}{t} = \frac{M_t\,\pi_m}{4\,\Omega^2 G\,t} \quad (\text{a } t \text{ costante})$$

con Ω area *racchiusa* dalla linea media e π_m suo perimetro. La prima discende dall'equilibrio alla torsione del flusso costante $q = \tau t$: $M_t = \oint q\, r\, \mathrm{d}s = q \cdot 2\Omega$.

Prompt pronto all'uso

Scrivi un programma Python sulla torsione di Bredt. Modello: sezione chiusa monocella in parete sottile, flusso di taglio costante, prima formula per τ e seconda per l'angolo unitario θ a spessore costante. Dati: cella rettangolare 600×200 mm, $t = 1{,}2$ mm, $M_t = 4{,}0$ kN·m, $G = 27$ GPa. Attenzione: Ω è l'area racchiusa, non l'area di materiale — dichiaralo in un commento. Formato: Ω, τ in MPa, θ in gradi/metro. Controllo: il flusso $q = \tau t$ deve valere $M_t/(2\Omega) = 16{,}67$ N/mm.

`bredt.py`

```python
# bredt.py -- torsione di un cassone monocella (formule di Bredt)
import math

a, b = 0.600, 0.200     # lati della linea media [m]
t    = 1.2e-3           # spessore uniforme [m]
Mt   = 4.0e3            # momento torcente [N m]
G    = 27e9             # modulo di taglio [Pa]

Omega = a*b                     # area RACCHIUSA dalla linea media
per   = 2*(a + b)               # perimetro della linea media
tau   = Mt/(2*Omega*t)                  # prima formula di Bredt
theta = Mt*per/(4*Omega**2*G*t)         # seconda formula [rad/m]

print(f"Area racchiusa   Omega = {Omega:.3f} m^2")
print(f"Flusso           q     = {tau*t/1000:6.2f} N/mm")
print(f"Tensione         tau   = {tau/1e6:6.2f} MPa")
print(f"Angolo unitario  theta = {math.degrees(theta):.3f} gradi/m")
```

Output

```
Area racchiusa   Omega = 0.120 m^2
Flusso           q     =  16.67 N/mm
Tensione         tau   =  13.89 MPa
Angolo unitario  theta = 0.196 gradi/m
```

Validazione. $q = M_t/(2\Omega) = 4000/0{,}24 = 16\,667$ N/m $= 16{,}67$ N/mm ✓, e $\tau = q/t = 16{,}67/1{,}2 = 13{,}89$ MPa ✓. Un cassone che ruota di due decimi di grado per metro sotto 4 kN m: è la sbalorditiva efficienza torsionale delle sezioni *chiuse*. Il confronto istruttivo è nella prova successiva.

Prova tu

Tagliate idealmente il cassone (sezione *aperta*): la teoria dà $\tau_{ap} = 3M_t/(\pi_m t^2)$ e rigidezza proporzionale a t^3. Calcolate il rapporto fra le tensioni chiusa/aperta e scoprite perché un portello non richiudibile con continuità strutturale è un problema serio, da compensare con robuste cornici.

11.6 Scheda 11.6 Verifica a taglio di una giunzione rivettata

Problema. Una giunzione deve trasmettere $F = 12$ kN con rivetti in 2117-T4 da $d = 4$ mm ($\tau_R = 240$ MPa, coefficiente di sicurezza $\nu = 1{,}5$). Determinare il numero minimo di rivetti a taglio semplice.

Formula chiave

Per n rivetti di diametro d a taglio semplice, con ripartizione uniforme:

$$\tau = \frac{F}{n \cdot \dfrac{\pi d^2}{4}} \le \tau_{amm} = \frac{\tau_R}{\nu} \quad \Longrightarrow \quad n \ge \frac{F}{\tau_{amm}\, \pi d^2/4}$$

Prompt pronto all'uso

Scrivi un programma Python per il dimensionamento di una giunzione rivettata. Modello: taglio semplice con ripartizione uniforme, ammissibile τ_R/ν, numero di rivetti arrotondato SEMPRE per eccesso con math.ceil. Dati: $F = 12$ kN, $d = 4$ mm, $\tau_R = 240$ MPa (2117-T4), $\nu = 1{,}5$. Formato: area del rivetto, forza ammissibile per rivetto, n, tensione effettiva a posteriori confrontata con l'ammissibile. Controllo: con questi dati $F/F_{riv} = 5{,}97$, quindi n deve valere 6, non 5.

giunzione_rivettata.py

```python
# giunzione_rivettata.py -- verifica a taglio di una fila di rivetti
import math

F       = 12e3          # forza da trasmettere [N]
d       = 4.0e-3        # diametro del rivetto [m]
tau_R   = 240e6         # tensione tangenziale di rottura (2117-T4) [Pa]
nu      = 1.5           # coefficiente di sicurezza

tau_amm = tau_R / nu
A_riv   = math.pi/4 * d**2
F_riv   = tau_amm * A_riv                 # forza ammissibile per rivetto
n       = math.ceil(F / F_riv)            # numero minimo di rivetti

```

```
print(f"Area del rivetto          : {A_riv*1e6:6.2f} mm^2")
print(f"Forza ammissibile/rivetto : {F_riv:6.0f} N")
print(f"Numero minimo di rivetti  : {n}")
print(f"Tensione effettiva        : {F/(n*A_riv)/1e6:6.1f} MPa "
      f"(ammissibile {tau_amm/1e6:.0f} MPa)")
```

Output

```
Area del rivetto          :  12.57 mm^2
Forza ammissibile/rivetto :   2011 N
Numero minimo di rivetti  : 6
Tensione effettiva        :  159.2 MPa (ammissibile 160 MPa)
```

Validazione. $F/F_{riv} = 12000/2011 = 5{,}97$: servono 6 rivetti, e infatti la tensione effettiva (159,2 MPa) scende appena sotto l'ammissibile. Notare `math.ceil`: l'arrotondamento è *sempre per eccesso* — con 5 rivetti la verifica non passerebbe. Un dettaglio da un carattere che vale una struttura.

Prova tu

Completate la verifica come nella prova d'esame: aggiungete il *rifollamento* della lamiera ($\sigma_{rif} = F/(n\,d\,t) \le \sigma_{rif,amm}$) e la verifica della sezione netta, e fate stampare al programma quale delle tre condizioni è dimensionante.

11.7 Scheda 11.7 L'asta di controventatura: carico critico

Problema. Un'asta di controventatura tubolare in lega leggera ($E = 72$ GPa, $D = 30$ mm, $t = 2$ mm, $L = 1{,}5$ m, cerniere agli estremi) lavora in compressione. Calcolare snellezza e carico critico euleriano, verificando la legittimità della formula.

Formula chiave

Per un'asta snella incernierata agli estremi:

$$P_{cr} = \frac{\pi^2 E I}{L^2}, \qquad \lambda = \frac{L}{\rho_g}, \quad \rho_g = \sqrt{I/A}$$

La formula vale in campo elastico, cioè per snellezze superiori alla snellezza limite del materiale ($\lambda \gtrsim 90$ per le leghe leggere): va *sempre* controllata.

Prompt pronto all'uso

Scrivi un programma Python sull'instabilità euleriana. Modello: $P_{cr} = \pi^2 EI/L^2$ per cerniera–cerniera; sezione anulare con $I = \pi(D^4 - d^4)/64$; calcola anche raggio giratore e snellezza e STAMPA un avviso se $\lambda < 90$ (formula fuori dominio). Dati: $E = 72$ GPa, tubo $D = 30$ mm, $t = 2$ mm, $L = 1{,}5$ m. Formato: I in mm⁴, A in mm², ρ_g in mm, λ, P_{cr} in kN. Controllo: I deve valere $\pi(30^4 - 26^4)/64 = 17329$ mm⁴.

`asta_eulero.py`

```
# asta_eulero.py -- carico critico di un'asta di controventatura
import math

```

```
E = 72e9              # modulo elastico lega di alluminio [Pa]
D = 0.030             # diametro esterno del tubo [m]
t = 0.002             # spessore [m]
L = 1.5               # lunghezza libera di inflessione [m]

d = D - 2*t
I = math.pi/64 * (D**4 - d**4)
A = math.pi/4  * (D**2 - d**2)
Pcr    = math.pi**2 * E * I / L**2        # formula di Eulero
rho_g  = math.sqrt(I/A)                   # raggio giratore
snell  = L / rho_g                        # snellezza

print(f"Inerzia          I    = {I*1e12:8.0f} mm^4")
print(f"Area             A    = {A*1e6:8.1f} mm^2")
print(f"Raggio giratore  rho  = {rho_g*1000:8.2f} mm")
print(f"Snellezza        lam  = {snell:8.1f}")
print(f"Carico critico   Pcr  = {Pcr/1000:8.2f} kN")
```

Output

```
Inerzia          I    =    17329 mm^4
Area             A    =    175.9 mm^2
Raggio giratore  rho  =     9.92 mm
Snellezza        lam  =    151.1
Carico critico   Pcr  =     5.47 kN
```

Validazione. $I = \pi/64\,(30^4 - 26^4) = \pi/64 \cdot 352976 = 17\,329\ \text{mm}^4$ ✓; la snellezza $151 \gg 90$ conferma la legittimità di Eulero; e $P_{cr} = \pi^2 \cdot 72 \cdot 10^9 \cdot 1{,}733 \cdot 10^{-8}/1{,}5^2 = 5{,}47\ \text{kN}$ ✓. Il programma controlla da solo la propria ipotesi di validità: è il livello di rigore che chiediamo in quinta.

Prova tu

A parità di area (quindi di massa), confrontate tubo e sezione piena: tabulate P_{cr} per D da 20 a 50 mm con A costante (ricavando t da A) e dimostrate numericamente perché le aste di controventatura sono *sempre* tubolari.

11.8 Scheda 11.8 La fusoliera pressurizzata

Problema. Una fusoliera cilindrica di raggio $r = 2$ m e spessore $t = 1{,}8$ mm opera con pressione differenziale di cabina $\Delta p = 0{,}060$ MPa. Calcolare le tensioni membranali e il margine di sicurezza (ammissibile 160 MPa).

Formula chiave

Per il cilindro in parete sottile (teoria membranale):

$$\sigma_c = \frac{\Delta p\, r}{t} \quad \text{(circonferenziale)}, \qquad \sigma_l = \frac{\Delta p\, r}{2t} \quad \text{(longitudinale)}$$

Dimostrazione della prima per equilibrio del semicilindro di lunghezza unitaria: $\Delta p \cdot 2r = 2\,\sigma_c\, t$. La circonferenziale è *doppia*: le eventuali fratture di un tubo in pressione corrono in direzione assiale, non trasversale.

Prompt pronto all'uso

Scrivi un programma Python sulle tensioni membranali di una fusoliera pressurizzata. Modello: cilindro in parete sottile, $\sigma_c = \Delta p\, r/t$ e $\sigma_l = \Delta p\, r/(2t)$; valido solo per $t \ll r$, controlla che $r/t > 20$. Dati: $r = 2$ m, $t = 1{,}8$ mm, $\Delta p = 0{,}060$ MPa, ammissibile 160 MPa. Formato: le due tensioni in MPa e il margine di sicurezza sulla peggiore. Controllo strutturale: la circonferenziale deve risultare esattamente doppia della longitudinale.

`fusoliera.py`

```
# fusoliera.py -- tensioni membranali di una fusoliera pressurizzata
dp  = 0.060e6      # pressione differenziale di cabina [Pa]
r   = 2.0          # raggio della fusoliera [m]
t   = 1.8e-3       # spessore del rivestimento [m]
amm = 160e6        # tensione ammissibile [Pa]

sigma_c = dp*r/t          # circonferenziale (cerchiante)
sigma_l = dp*r/(2*t)      # longitudinale

print(f"Rapporto r/t              : {r/t:6.0f}  (parete sottile)")
print(f"Tensione circonferenziale : {sigma_c/1e6:6.1f} MPa")
print(f"Tensione longitudinale    : {sigma_l/1e6:6.1f} MPa")
print(f"Margine di sicurezza      : {amm/sigma_c - 1:6.2f}")
```

Output

```
Rapporto r/t              :   1111  (parete sottile)
Tensione circonferenziale :   66.7 MPa
Tensione longitudinale    :   33.3 MPa
Margine di sicurezza      :   1.40
```

Validazione. $\sigma_c = 60000 \cdot 2/0{,}0018 = 66{,}7$ MPa ✓, ed è esattamente doppia della longitudinale come impone la statica — il controllo strutturale del prompt. Il margine 1,40 sembra generoso, ma il dimensionamento reale della fusoliera è governato dalla *fatica*: la cabina si pressurizza e depressurizza a ogni volo, e la lezione dei Comet del 1954 — cedimenti per fatica innescati agli spigoli dei finestrini — è il motivo per cui i finestrini che vedete oggi sono ovali.

Prova tu

Tabulate σ_c al variare della quota: Δp è la differenza fra la pressione di cabina (ISA a 2400 m, importate `atmosfera20.py`) e quella esterna ISA alla quota di volo, da 6000 a 12 000 m. A quale quota si raggiunge la Δp di progetto?

11.9 Scheda 11.9 Moto isoentropico: rapporti di ristagno e area critica

Problema. Per quattro valori del numero di Mach (0,5; 0,8; 1; 2) calcolare i rapporti T_0/T, p_0/p e A/A^* del moto isoentropico monodimensionale ($\gamma = 1{,}4$), ricostruendo la tabella che troverete su ogni testo di gasdinamica.

Formula chiave

$$\frac{T_0}{T} = 1 + \frac{\gamma-1}{2}M^2, \qquad \frac{p_0}{p} = \left(\frac{T_0}{T}\right)^{\frac{\gamma}{\gamma-1}}, \qquad \frac{A}{A^*} = \frac{1}{M}\left[\frac{2}{\gamma+1}\left(1+\frac{\gamma-1}{2}M^2\right)\right]^{\frac{\gamma+1}{2(\gamma-1)}}$$

A^* è l'area della sezione (reale o virtuale) in cui $M = 1$: il minimo di A/A^* vale 1 proprio in gola — il cuore del convergente–divergente.

Prompt pronto all'uso

Scrivi un programma Python per il moto isoentropico monodimensionale. Modello: le tre relazioni di ristagno e d'area con $\gamma = 1{,}4$; aggiungi i rapporti critici p^*/p_0 e T^*/T_0. Dati: $M = 0{,}5;\ 0{,}8;\ 1{,}0;\ 2{,}0$. Formato: tabella a quattro decimali. Controlli canonici: a $M = 1$ deve risultare $A/A^* = 1$ esatto; p^*/p_0 deve valere 0,5283 e T^*/T_0 esattamente $2/(\gamma+1) = 0{,}8333$.

`isoentropico.py`

```python
# isoentropico.py -- rapporti del moto isoentropico (gamma = 1.4)
g = 1.4

print("  M     T0/T     p0/p      A/A*")
for M in [0.5, 0.8, 1.0, 2.0]:
    TT = 1 + (g-1)/2*M**2
    pp = TT**(g/(g-1))
    AA = 1/M * ((2/(g+1))*TT)**((g+1)/(2*(g-1)))
    print(f" {M:3.1f}   {TT:.4f}   {pp:.4f}    {AA:.4f}")

print(f"\nRapporti critici:  p*/p0 = {(2/(g+1))**(g/(g-1)):.4f}"
      f"   T*/T0 = {2/(g+1):.4f}")
```

Output

```
M     T0/T     p0/p      A/A*
 0.5   1.0500   1.1862    1.3398
 0.8   1.1280   1.5243    1.0382
 1.0   1.2000   1.8929    1.0000
 2.0   1.8000   7.8244    1.6875

Rapporti critici:  p*/p0 = 0.5283   T*/T0 = 0.8333
```

Validazione. I tre controlli canonici sono soddisfatti: $A/A^* = 1$ esatto a $M = 1$, $p^*/p_0 = 0{,}5283$ e $T^*/T_0 = 0{,}8333$ ✓ — gli stessi valori delle tabelle di ogni manuale di gasdinamica, che ora sapete *rigenerare* anziché consultare. Osservate $M = 0{,}8$: $p_0/p = 1{,}52$, mentre il Bernoulli incomprimibile darebbe $1+\gamma M^2/2 = 1{,}448$: il 5% di scarto è la misura dell'errore del Pitot non corretto in alto subsonico, promessa mantenuta della scheda 9.7.

Prova tu

Invertite il problema con la bisezione della scheda 9.4: dato $A/A^* = 1{,}6875$, trovate i *due* Mach che lo producono (uno subsonico e uno supersonico: scegliete l'intervallo di ricerca!). È il funzionamento fuori progetto dell'ugello, e il motivo per cui la stessa geometria può contenere due regimi.

11.10 Scheda 11.10 L'onda d'urto normale

Problema. Calcolare i salti di tutte le grandezze attraverso un urto normale per un Mach a monte assegnato ($M_1 = 2$), inclusa la perdita di pressione totale.

Formula chiave

Attraverso un urto normale in gas perfetto ($\gamma = 1{,}4$):

$$M_2^2 = \frac{1 + \frac{\gamma-1}{2} M_1^2}{\gamma M_1^2 - \frac{\gamma-1}{2}}, \qquad \frac{p_2}{p_1} = 1 + \frac{2\gamma}{\gamma+1}(M_1^2 - 1), \qquad \frac{\rho_2}{\rho_1} = \frac{(\gamma+1)M_1^2}{(\gamma-1)M_1^2 + 2}$$

Prompt pronto all'uso

Scrivi un programma Python sull'urto normale. Modello: relazioni di Rankine–Hugoniot per gas perfetto, $\gamma = 1{,}4$; includi M_2, p_2/p_1, ρ_2/ρ_1, T_2/T_1 (come rapporto dei primi due) e la perdita di pressione totale p_{02}/p_{01}. Dati: M_1 da input, con controllo $M_1 > 1$ e messaggio esplicativo. Formato: quattro decimali. Controlli canonici a $M_1 = 2$: $M_2 = 0{,}5774$, $p_2/p_1 = 4{,}5$ esatto, $\rho_2/\rho_1 = 8/3$, $p_{02}/p_{01} = 0{,}7209$.

urto_normale.py

```
# urto_normale.py -- salti attraverso un'onda d'urto normale
import math
g = 1.4                                        # gamma dell'aria

M1 = float(input("Mach a monte M1 (>1): "))
M2    = math.sqrt((1 + (g-1)/2*M1**2) / (g*M1**2 - (g-1)/2))
p2p1  = 1 + 2*g/(g+1) * (M1**2 - 1)
r2r1  = (g+1)*M1**2 / ((g-1)*M1**2 + 2)
T2T1  = p2p1 / r2r1
pt    = (((g+1)*M1**2/((g-1)*M1**2+2))**(g/(g-1))
         * ((g+1)/(2*g*M1**2-(g-1)))**(1/(g-1)))

print(f"M2        = {M2:.4f}")
print(f"p2/p1     = {p2p1:.3f}")
print(f"rho2/rho1 = {r2r1:.3f}")
print(f"T2/T1     = {T2T1:.4f}")
print(f"p02/p01   = {pt:.4f}   (perdita di pressione totale)")
```

Output

```
Mach a monte M1 (>1): 2.0
M2        = 0.5774
p2/p1     = 4.500
rho2/rho1 = 2.667
T2/T1     = 1.6875
p02/p01   = 0.7209   (perdita di pressione totale)
```

Validazione. I valori a $M_1 = 2$ sono canonici e tabellati in ogni testo di gasdinamica: $M_2 = 0{,}5774$, $p_2/p_1 = 4{,}5$ esatto ($1 + \frac{2{,}8}{2{,}4} \cdot 3 = 1 + 3{,}5$), $\rho_2/\rho_1 = 8/3$, $p_{02}/p_{01} = 0{,}7209$ ✓. La perdita del 28% di pressione totale misura l'irreversibilità dell'urto: è il prezzo entropico del volo supersonico.

Prova tu

Aggiungete il controllo $M_1 > 1$ richiesto dal prompt, poi tracciate p_2/p_1 e p_{02}/p_{01} per M_1 da 1 a 5: il grafico mostra perché le prese d'aria supersoniche evitano l'urto normale unico a favore di urti obliqui multipli.

Verso la seconda prova

Le dieci schede di questo capitolo coprono i mattoni ricorrenti delle prove ministeriali (longherone, sezioni, flussi di taglio, diagrammi lungo l'ala, lungo l'ala, torsione, giunzioni, aste, pressurizzazione, comprimibile). Il metodo di lavoro per l'esame è questo: risolvete la traccia *a mano*, come richiesto in sede d'esame, e usate i vostri programmi come *banco di collaudo* della soluzione manuale — non il contrario. Chi ha costruito e validato i propri strumenti durante l'anno arriva alla prova con un vantaggio che nessuna IA può dare all'ultimo momento: sa *riconoscere* un risultato sbagliato a colpo d'occhio, perché ne ha visti mille giusti.

Parte V

Corso avanzato

Introduzione alla fluidodinamica computazionale, per chi proseguirà gli studi

Capitolo 12

Verso la CFD: equazioni di governo e differenze finite

Nota

Questa parte del volume è un *corso avanzato*, pensato per chi proseguirà gli studi in ingegneria: non è materia d'esame di Stato, ma il ponte fra ciò che avete costruito nelle parti precedenti e la fluidodinamica computazionale che incontrerete all'università. Il percorso è organizzato in **dodici esercitazioni** a difficoltà crescente: si parte da un'equazione a una derivata e si arriva, gradino per gradino, a risolvere le equazioni di Navier–Stokes su due problemi canonici. Tutto il bagaglio necessario — NumPy, Matplotlib, il ciclo GLVC, la disciplina della validazione — lo possedete già.

12.1 Che cos'è la fluidodinamica computazionale

Delle equazioni che governano il moto dei fluidi si conoscono soluzioni esatte solo per una manciata di casi idealizzati: il profilo di Poiseuille in un condotto, il flusso di Couette, poche altre geometrie accademiche. Attorno a un'ala vera, dentro una presa d'aria, nella scia di una fusoliera, la soluzione esatta non esiste: esistono la galleria del vento e il calcolatore. La *fluidodinamica computazionale* (CFD, *Computational Fluid Dynamics*) è la disciplina che sostituisce alle derivate delle equazioni di governo delle differenze fra valori su una griglia di punti, trasformando un problema differenziale — infinite incognite — in un problema algebrico con un numero finito di incognite, che il calcolatore sa affrontare.

Il prezzo di questa trasformazione è il tema centrale di tutta la parte: la soluzione numerica *non è* la soluzione esatta. È un'approssimazione, con errori che vanno capiti, misurati e tenuti sotto controllo. Un ingegnere che usa un codice CFD senza sapere che cosa c'è dentro è nella stessa posizione di chi accetta il codice di un'IA senza collaudarlo: il metodo GLVC del capitolo 7 vale identico, con la fisica al posto del valore di controllo.

12.2 Le equazioni di governo

Per un fluido incomprimibile a viscosità costante, le equazioni di Navier–Stokes esprimono due principi che già conoscete: la conservazione della massa e la seconda legge della dinamica applicata a una particella fluida.

Formula chiave

$$\nabla \cdot \mathbf{V} = 0 \qquad \text{(continuità)}$$

$$\frac{\partial \mathbf{V}}{\partial t} + (\mathbf{V} \cdot \nabla)\mathbf{V} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{V} \qquad \text{(quantità di moto)}$$

A sinistra: la variazione nel tempo più il trasporto *convettivo*, non lineare perché la velocità trasporta se stessa. A destra: il gradiente di pressione e la diffusione *viscosa*, con $\nu = \mu/\rho$ viscosità cinematica.

Nel piano, con $\mathbf{V} = (u, v)$, il sistema si scrive per esteso:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \qquad \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

e analogamente per v. Tre incognite (u, v, p), tre equazioni, nessuna soluzione generale nota: è il problema che affronteremo nell'ultima esercitazione.

Osservazione

La strategia didattica di tutto il percorso è *smontare* questo sistema nei suoi termini elementari e studiarli uno alla volta con *equazioni modello*: prima il solo trasporto (convezione), poi la sola viscosità (diffusione), poi i due insieme (equazione di Burgers), poi la pressione (equazioni di Laplace e Poisson). Quando ogni pezzo sarà stato capito e collaudato separatamente, rimontarli in Navier–Stokes sarà un assemblaggio di componenti già validati — esattamente come si costruisce un velivolo.

12.3 Dal continuo alla griglia: le differenze finite

Il fondamento di tutto è lo sviluppo di Taylor. Se $u(x)$ è regolare,

$$u(x + \Delta x) = u(x) + \Delta x\, u'(x) + \frac{\Delta x^2}{2}u''(x) + \frac{\Delta x^3}{6}u'''(x) + \dots$$

Risolvendo rispetto a $u'(x)$ si ottengono le tre approssimazioni fondamentali della derivata prima, ciascuna con il proprio *errore di troncamento*:

Formula chiave

$$\underbrace{u_i' \simeq \frac{u_{i+1} - u_i}{\Delta x}}_{\text{in avanti, } O(\Delta x)} \qquad \underbrace{u_i' \simeq \frac{u_i - u_{i-1}}{\Delta x}}_{\text{all'indietro, } O(\Delta x)} \qquad \underbrace{u_i' \simeq \frac{u_{i+1} - u_{i-1}}{2\Delta x}}_{\text{centrata, } O(\Delta x^2)}$$

e per la derivata seconda, sommando i due sviluppi in $\pm\Delta x$:

$$u_i'' \simeq \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} \qquad O(\Delta x^2)$$

La notazione $O(\Delta x^n)$ è una promessa verificabile: dimezzando il passo, l'errore di uno schema del primo ordine si dimezza, quello di uno schema del secondo ordine si riduce di *quattro* volte. Nelle esercitazioni misureremo entrambe le cose, perché la verifica dell'ordine di convergenza è il collaudo più severo che un codice numerico possa superare.

Osservazione

La *dimostrazione* dell'ordine della differenza centrata è un'ottima palestra: sottraendo lo sviluppo di $u(x-\Delta x)$ da quello di $u(x+\Delta x)$, i termini pari (u, u'') si elidono e resta $u_{i+1} - u_{i-1} = 2\Delta x\, u' + \frac{\Delta x^3}{3}u''' + \dots$: dividendo per $2\Delta x$, l'errore dominante è $\frac{\Delta x^2}{6}u'''$, del secondo ordine. Due righe di algebra che spiegano un comportamento numerico che vedrete coi vostri occhi.

12.4 Griglia, condizioni iniziali e al contorno

Discretizzare significa sostituire al dominio continuo una griglia di n_x punti a passo Δx (in due dimensioni, $n_x \times n_y$ punti) e alla funzione $u(x, t)$ la tabella dei valori u_i^n, dove i conta i punti e n i passi temporali di ampiezza

Δt. In NumPy la griglia è `np.linspace`, la soluzione è un array e l'avanzamento nel tempo è un ciclo `for` che aggiorna l'array: tutto già visto nella Parte I.

Un problema di evoluzione richiede sempre due ingredienti oltre all'equazione: la *condizione iniziale* (lo stato a $t = 0$) e le *condizioni al contorno* (che cosa accade ai bordi del dominio). In queste esercitazioni useremo condizioni di Dirichlet (valore assegnato al bordo), di Neumann (derivata assegnata) e periodiche; la scelta della condizione iniziale non sarà mai casuale: quasi sempre sceglieremo profili — impulsi gaussiani, fronti viaggianti, soluzioni manifatturate — per i quali la soluzione *esatta* è nota, così che ogni esercitazione porti incorporato il proprio collaudo. È la regola aurea del volume applicata alla CFD.

12.5 Stabilità: un'anticipazione

C'è un fenomeno, tipico dei metodi espliciti, che non ha equivalente nel calcolo manuale: scegliendo un passo temporale troppo grande rispetto al passo spaziale, la soluzione numerica *esplode* — oscillazioni che si amplificano a ogni passo fino all'overflow. Non è un errore di programmazione: è un'*instabilità numerica*, e la condizione che la governa (il numero di Courant) sarà l'oggetto di un interludio dedicato, subito dopo le prime due esercitazioni. L'anticipiamo qui perché è la prima cosa da sospettare quando un calcolo CFD produce `nan` o numeri con esponente a tre cifre.

12.6 La mappa del percorso

#	Esercitazione	Termine studiato	Collaudo
1	Convezione lineare 1D	trasporto	traslazione esatta
2	Convezione non lineare 1D	$u\,\partial u/\partial x$	irripidimento
–	*Interludio*: CFL	stabilità	esplosione controllata
3	Diffusione 1D	$\nu\,\partial^2 u/\partial x^2$	picco $\to 1/\sqrt{2}$
4	Burgers 1D	convezione + diffusione	fronte esatto
–	*Interludio*: vettorizzazione	prestazioni	stesso risultato
5	Convezione lineare 2D	trasporto piano	traslazione diagonale
6	Convezione non lineare 2D	sistema accoppiato	simmetria $u \leftrightarrow v$
7	Diffusione 2D	Laplaciano	picco $\to 1/2$
8	Burgers 2D	tutto insieme	simmetria
9	Laplace 2D	pressione (omogenea)	soluzione in serie
10	Poisson 2D	pressione con sorgente	sol. manifatturata
11	Cavità con coperchio	Navier–Stokes	$r \cdot \mathbf{V} \simeq 0$
12	Canale con forzante	Navier–Stokes	parabola di Poiseuille

Ogni esercitazione ha la struttura delle schede annuali: equazione e discretizzazione, eventuale prompt per l'IA, codice, output autentico, validazione. La colonna di destra della tabella è la promessa che ciascuna dovrà mantenere.

Capitolo 13

Le equazioni modello in una dimensione

Nota

Quattro esercitazioni e due interludi. Le condizioni iniziali sono scelte perché la soluzione esatta sia nota: l'impulso gaussiano per la convezione e la diffusione, il fronte viaggiante per Burgers. Ogni numero stampato proviene da esecuzione reale.

13.1 Esercitazione 1 Convezione lineare monodimensionale

Il primo termine da isolare è il trasporto. L'equazione modello è

$$\frac{\partial u}{\partial t} + c\,\frac{\partial u}{\partial x} = 0$$

la cui soluzione esatta è nota per intero: qualunque profilo iniziale $u_0(x)$ viene *traslato rigidamente* a velocità c, cioè $u(x,t) = u_0(x - ct)$. È l'idealizzazione di una perturbazione di pressione trasportata dalla corrente. Proprio perché la soluzione esatta è banale, questa è l'equazione perfetta per il primo collaudo di un metodo numerico: ogni deformazione del profilo è errore numerico, riconoscibile a vista.

Formula chiave

Discretizzazione con differenza in avanti nel tempo e all'indietro nello spazio (schema *upwind*, legittimo per $c > 0$: le informazioni arrivano da sinistra, da monte):

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\,\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0 \quad\Longrightarrow\quad u_i^{n+1} = u_i^n - \frac{c\,\Delta t}{\Delta x}\,(u_i^n - u_{i-1}^n)$$

Il gruppo adimensionale $\sigma = c\,\Delta t/\Delta x$ è il *numero di Courant*: ne riparleremo presto.

Prompt pronto all'uso

Scrivi un programma Python con NumPy per la convezione lineare 1D. Modello: schema upwind esplicito $u_i^{n+1} = u_i^n - \sigma(u_i^n - u_{i-1}^n)$ con $c = 1$; aggiorna l'array con lo slicing u[1:], senza cicli sui punti. Dati: dominio $[0, 2]$ con 201 punti, impulso gaussiano $u_0 = \exp[-((x - 0{,}3)/0{,}1)^2]$, $\sigma = 0{,}5$, 100 passi (tempo finale 0,5). Formato: passo, dt, tempo finale, picco e sua posizione. Controllo: la soluzione esatta è il profilo iniziale traslato in $x = 0{,}8$; confronta e stampa l'errore massimo. Il picco numerico sarà minore di 1: commenta il perché in una riga.

convezione1d.py

```
# convezione1d.py -- trasporto di un impulso con schema upwind
import numpy as np

c, L, nx = 1.0, 2.0, 201
x = np.linspace(0, L, nx); dx = x[1] - x[0]
sigma = 0.5; dt = sigma*dx/c; nt = 100          # t finale = 0.5

```

```
u = np.exp(-((x - 0.3)/0.1)**2)                  # impulso gaussiano
for _ in range(nt):
    u[1:] = u[1:] - c*dt/dx*(u[1:] - u[:-1])     # upwind (c > 0)
    u[0] = 0.0                                   # ingresso indisturbato

t = nt*dt
u_es = np.exp(-((x - 0.3 - c*t)/0.1)**2)         # esatta: traslazione
print(f"dx = {dx:.4f}  dt = {dt:.4f}  sigma = {c*dt/dx:.2f}  t = {t:.2f}")
print(f"picco numerico {u.max():.4f} in x = {x[u.argmax()]:.3f}"
      f"   (esatto: 1.0000 in x = 0.800)")
print(f"errore massimo |u - u_es| = {np.abs(u - u_es).max():.4f}")
```

Output

```
dx = 0.0100  dt = 0.0050  sigma = 0.50  t = 0.50
picco numerico 0.8163 in x = 0.800   (esatto: 1.0000 in x = 0.800)
errore massimo |u - u_es| = 0.1837
```

Validazione. Il picco arriva *esattamente* dove deve (la velocità di trasporto è riprodotta senza errore), ma la sua altezza è scesa da 1 a 0,82: lo schema upwind, del primo ordine, introduce una *diffusione numerica* che smussa il profilo come farebbe una viscosità fittizia. Non è un difetto di programmazione: è la firma dell'errore di troncamento $O(\Delta x)$, e l'esercizio seguente ne misura la legge.

Prova tu

Raddoppiate la risoluzione ($n_x = 401$, mantenendo $\sigma = 0{,}5$) e poi raddoppiatela ancora: l'errore massimo deve circa dimezzarsi a ogni raffinamento — è la promessa del primo ordine, verificatela. Poi sostituite l'impulso con un gradino: quale dei due profili soffre di più la diffusione numerica, e perché?

13.2 Esercitazione 2 Convezione non lineare

Nelle equazioni vere la velocità trasporta *se stessa*: il termine convettivo è $u\,\partial u/\partial x$. L'equazione modello diventa

$$\frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial x} = 0 \qquad \Longrightarrow \qquad u_i^{n+1} = u_i^n - u_i^n\,\frac{\Delta t}{\Delta x}\,(u_i^n - u_{i-1}^n)$$

e la fisica cambia qualitativamente: ogni punto del profilo viaggia alla *propria* velocità. La cresta dell'impulso ($u = 2$) corre il doppio della base ($u = 1$): il fronte anteriore si impenna, quello posteriore si distende. È il meccanismo che, portato al limite, genera l'onda d'urto della scheda II.10.

convezione_nl.py (parte centrale)

```
u = 1.0 + np.exp(-((x - 0.3)/0.1)**2)      # base 1, cresta 2
pend0 = np.abs(np.gradient(u, x)).max()    # pendenza massima iniziale
dt = 0.5*dx/2.0                            # CFL con u_max = 2
for _ in range(int(0.10/dt)):
    u[1:] = u[1:] - u[1:]*dt/dx*(u[1:] - u[:-1])
pend1 = np.abs(np.gradient(u, x)).max()
print(f"t = 0.100  pendenza massima: {pend0:.2f} -> {pend1:.2f}"
      f"  (irripidimento {pend1/pend0:.2f}x)")
```

```
Output
```

```
t = 0.100  pendenza massima: 8.52 -> 14.83  (irripidimento 1.74x)
```

Validazione. La teoria delle caratteristiche prevede che la pendenza massima diverga al tempo $t_s = 1/\max|u_0'| = 1/8{,}52 = 0{,}117$: a $t = 0{,}10$, ossia all'85% del tempo di formazione dell'urto, il fronte si è già irripidito del 74%. Notare il passo temporale: la condizione di stabilità va imposta sulla velocità *massima* del campo, non su quella media — il primo esempio di come la non linearità complichi la vita al numerico.

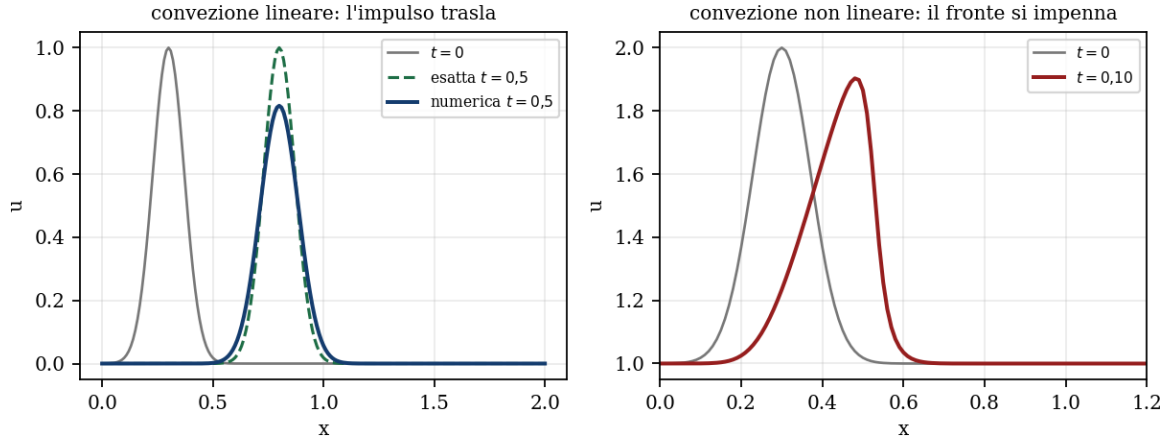


Figura 13.1. A sinistra la convezione lineare: l'impulso trasla e la diffusione numerica lo smussa. A destra la non lineare: la cresta corre più della base e il fronte si impenna.

Interludio: la condizione di stabilità CFL

Che cosa succede raffinando la griglia *senza* ridurre il passo temporale? L'esperimento è più eloquente di ogni teorema: stessa equazione, stesso $\Delta t = 0{,}005$, tre griglie via via più fitte.

```
Output
```

```
nx= 201  dx=0.0100  sigma=0.500   max|u| = 8.163e-01
nx= 335  dx=0.0060  sigma=0.835   max|u| = 9.536e-01
nx= 501  dx=0.0040  sigma=1.250   max|u| = 1.484e+12
```

Con $\sigma = 1{,}25$ la soluzione non è «meno accurata»: è *esplosa*, dodici ordini di grandezza in cento passi. Riducendo invece Δt insieme a Δx, così da mantenere $\sigma = 0{,}5$, il calcolo resta stabile su ogni griglia e l'errore *diminuisce* raffinando ($\max|u| = 0{,}816,\ 0{,}913,\ 0{,}954$: il picco risale verso il valore esatto 1).

Formula chiave

Condizione di Courant–Friedrichs–Lewy (CFL) per lo schema upwind esplicito:

$$\sigma = \frac{c\,\Delta t}{\Delta x} \leq 1$$

Interpretazione fisica: in un passo temporale, l'informazione fisica percorre $c\,\Delta t$; lo schema numerico «vede» solo fino al punto adiacente, a distanza Δx. Se l'informazione fisica corre più del dominio di dipendenza numerico ($\sigma > 1$), lo schema pretende di calcolare il futuro senza i dati che lo determinano — e la matematica si vendica con l'instabilità.

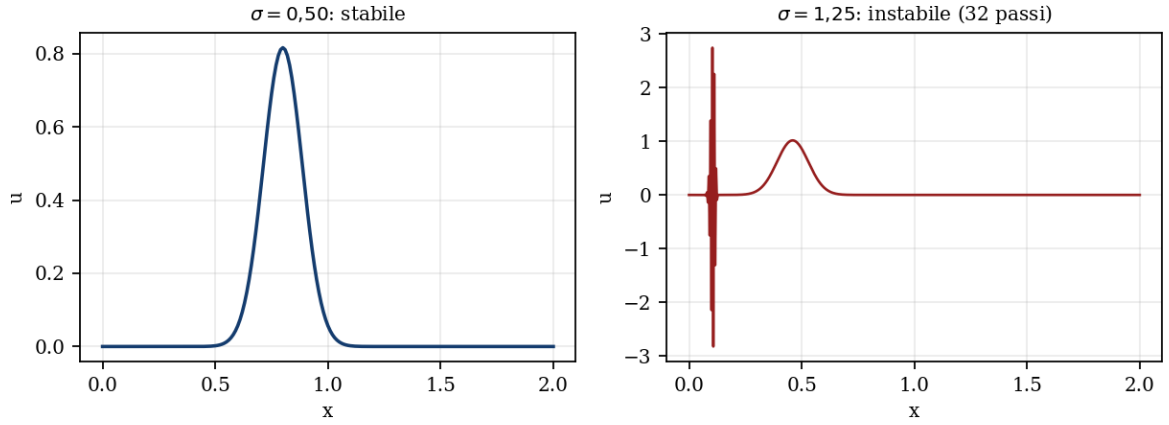


Figura 13.2. La stessa equazione, lo stesso schema: con $\sigma = 0{,}5$ la soluzione è sana; con $\sigma = 1{,}25$ bastano 32 passi per generare oscillazioni di ampiezza crescente.

Osservazione

La lezione operativa: quando un calcolo produce `nan`, `inf` o numeri a esponente enorme, il primo sospettato è *sempre* il passo temporale, non il codice. Riducetelo di un fattore 10: se l'esplosione sparisce, avete diagnosticato un'instabilità, e la cura è legare Δt a Δx tramite σ, mai fissarlo alla cieca.

13.3 Esercitazione 3 Diffusione

Secondo ingrediente di Navier–Stokes: la viscosità. Da sola governa l'equazione della diffusione,

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} \qquad \Longrightarrow \qquad u_i^{n+1} = u_i^n + r\,(u_{i+1}^n - 2u_i^n + u_{i-1}^n), \quad r = \frac{\nu\,\Delta t}{\Delta x^2}$$

con la derivata seconda discretizzata dalla differenza centrata a tre punti. La stabilità ora richiede $r \le 1/2$: si noti il Δx^2 a denominatore, che rende la diffusione esplicita *molto* più esigente della convezione sul passo temporale quando si raffina la griglia.

La condizione iniziale gaussiana è di nuovo una scelta strategica: la soluzione esatta dell'equazione del calore per un profilo gaussiano *resta gaussiana*, con varianza che cresce linearmente nel tempo, $s^2(t) = s_0^2 + 2\nu t$, e picco che scende come $s_0/s(t)$. Scegliendo $t = s_0^2/(2\nu)$ la varianza *raddoppia* e il picco deve valere esattamente $1/\sqrt{2}$.

`diffusione1d.py` (parte centrale)

```
nu, s0 = 0.05, 0.1
u = np.exp(-(x - 1.0)**2/(2*s0**2))
dt = 4e-4; r = nu*dt/dx**2; nt = 250          # t = 0.1 = s0^2/(2 nu)
for _ in range(nt):
    u[1:-1] = u[1:-1] + r*(u[2:] - 2*u[1:-1] + u[:-2])
picco_es = s0/np.sqrt(s0**2 + 2*nu*nt*dt)
print(f"r = {r:.3f} (< 0.5: stabile)   t = {nt*dt:.2f}")
print(f"picco numerico {u.max():.4f}   esatto {picco_es:.4f} = 1/sqrt(2)")
```

```
Output
```

```
r = 0.200 (< 0.5: stabile)   t = 0.10
picco numerico 0.7071   esatto 0.7071 = 1/sqrt(2)
```

Validazione. Quattro cifre esatte contro la soluzione analitica: la differenza centrata è del secondo ordine e su questo profilo liscio l'errore è invisibile alla quarta cifra. Confrontate con la convezione dell'esercitazione 13.1: là il primo ordine perdeva il 18% del picco, qui il secondo ordine non perde nulla di misurabile. L'ordine dello schema non è pedanteria: è la differenza fra un risultato utilizzabile e uno da buttare.

Prova tu

Provate $r = 0{,}55$: l'instabilità della diffusione ha una firma diversa da quella della convezione — oscillazioni a dente di sega punto-per-punto (la componente a lunghezza d'onda $2\Delta x$ è la prima ad amplificarsi). Imparate a riconoscerla: nei codici veri è il sintomo che distingue «passo troppo grande per la viscosità» da «passo troppo grande per la convezione».

13.4 Esercitazione 4 L'equazione di Burgers: nasce il fronte viscoso

Convezione non lineare più diffusione: l'equazione di Burgers,

$$\frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial x} = \nu\,\frac{\partial^2 u}{\partial x^2}$$

è il modello ridotto più fedele di Navier–Stokes: contiene la stessa competizione fra l'irripidimento convettivo e lo smussamento viscoso. Il loro equilibrio produce la soluzione esatta più istruttiva di tutto il percorso, il *fronte viaggiante*:

Formula chiave

$$u(x,t) = \frac{u_L + u_R}{2} - \frac{u_L - u_R}{2}\,\tanh\!\left[\frac{(u_L - u_R)\,(x - s t - x_0)}{4\nu}\right], \qquad s = \frac{u_L + u_R}{2}$$

Un fronte che collega i livelli u_L (a monte) e u_R (a valle), viaggia alla velocità media s e mantiene *per sempre* lo spessore $\delta \sim 4\nu/(u_L - u_R)$: la convezione che vorrebbe irripidirlo e la viscosità che vorrebbe distenderlo si equilibrano esattamente. È il ritratto in miniatura dell'onda d'urto reale, il cui spessore — pochi liberi cammini medi — nasce dallo stesso bilancio.

```
burgers1d.py
```

```
# burgers1d.py -- fronte viscoso viaggiante: numerico vs esatto
import numpy as np

nu = 0.05; uL, uR = 2.0, 0.0; s = (uL + uR)/2
nx = 801; x = np.linspace(0, 4, nx); dx = x[1] - x[0]

def esatta(x, t):
    return s - (uL-uR)/2*np.tanh((uL-uR)*(x - 1.0 - s*t)/(4*nu))

u = esatta(x, 0.0)                      # partiamo dalla soluzione esatta
dt = 0.2*dx**2/nu                       # r = 0.2 (domina la diffusione)
nt = int(round(1.0/dt))                 # tempo finale t = 1
for _ in range(nt):
    un = u.copy()
```

```
    u[1:-1] = (un[1:-1] - un[1:-1]*dt/dx*(un[1:-1] - un[:-2])
               + nu*dt/dx**2*(un[2:] - 2*un[1:-1] + un[:-2]))
    u[0], u[-1] = uL, uR

t = nt*dt
print(f"dx = {dx:.5f}  dt = {dt:.2e}  t = {t:.1f}")
print(f"fronte numerico (u=1) in x = {x[np.argmin(np.abs(u-1.0))]:.3f}"
      f"   (esatto: 2.000)")
print(f"errore massimo = {np.abs(u - esatta(x, t)).max():.4f}")
print(f"spessore teorico del fronte: {4*nu/(uL-uR):.3f}")
```

```
Output

dx = 0.00500  dt = 1.00e-04  t = 1.0
fronte numerico (u=1) in x = 1.985   (esatto: 2.000)
errore massimo = 0.1503
spessore teorico del fronte: 0.100
```

Validazione (di convergenza, la più severa). L'errore 0,15 pare grande, ma va letto insieme alla pendenza: sul fronte $|u'| = 10$, e un ritardo di fase di 15 millesimi basta a produrlo. La prova decisiva è rieseguire su tre griglie:

```
Output

dx = 0.01000   err_max = 0.2829
dx = 0.00500   err_max = 0.1503
dx = 0.00250   err_max = 0.0777
```

L'errore si *dimezza* a ogni dimezzamento del passo: primo ordine, esattamente come promesso dallo schema upwind sul termine convettivo (che qui domina l'errore, sovrastando il secondo ordine del termine viscoso). Un codice di cui si conosce e si misura l'ordine di convergenza è un codice *verificato*: è il criterio con cui, all'università e in azienda, si giudica ogni solutore.

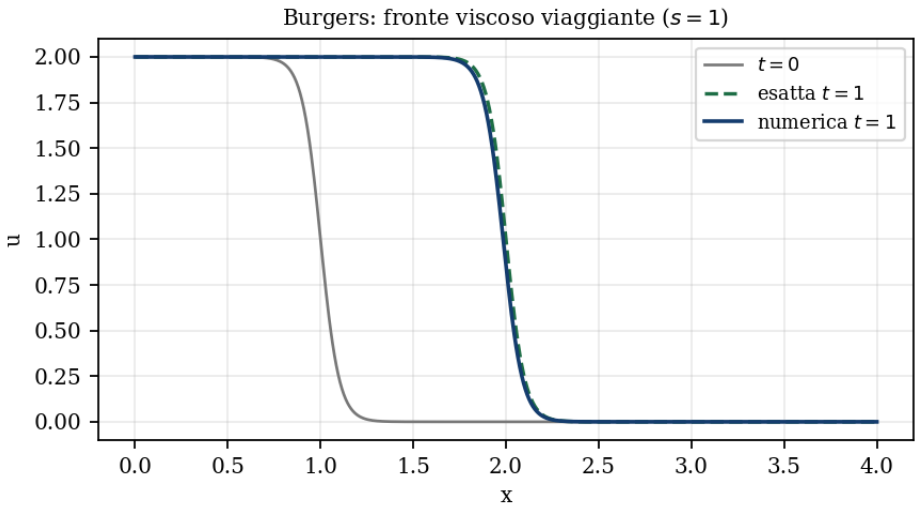


Figura 13.3. Il fronte di Burgers dopo un secondo di viaggio: la soluzione numerica insegue quella esatta con un lieve ritardo di fase, prezzo del primo ordine.

Interludio: la vettorizzazione

Tutti i codici del capitolo aggiornano gli array con lo *slicing* (`u[1:] - u[:-1]`) anziché con un ciclo sui punti. Non è vezzo stilistico. Un solo passo di convezione su 200 001 punti, misurato con `time.perf_counter`:

```
Output
```

```
un passo, nx = 200001:  ciclo 150.6 ms,  slicing 31.22 ms
accelerazione 5x   differenza massima fra i risultati: 0.0e+00
```

Cinque volte più veloce, risultato *bit per bit* identico. Su un caso 2D con centinaia di migliaia di passi, il fattore 5 (che su operazioni più ricche diventa 50 o 500) è la differenza fra un calcolo da minuti e uno da ore. La regola: in NumPy il ciclo `for` si usa per avanzare nel *tempo*, mai per scorrere i punti dello *spazio*.

Dialoga con l'IA

Prendete il vostro `burgers1d.py` e chiedete all'assistente: *"Riscrivi il ciclo temporale sostituendo lo schema upwind sul termine convettivo con una differenza centrata"*. Eseguite: il calcolo sviluppa oscillazioni a monte del fronte. Chiedete poi il perché, e confrontate la spiegazione con quanto sapete dalla condizione CFL: la differenza centrata pura sulla convezione è *incondizionatamente* instabile con l'avanzamento di Eulero esplicito — un esempio da manuale di risposta dell'IA da validare con la teoria, non con la fiducia.

Capitolo 14

Due dimensioni ed equazioni ellittiche

Nota

Sei esercitazioni. Le prime quattro estendono al piano le equazioni modello del capitolo precedente: la novità è tecnica (array bidimensionali, slicing su due indici), non concettuale. Le ultime due introducono una famiglia nuova, le equazioni *ellittiche*, che non evolvono nel tempo ma impongono un equilibrio istantaneo su tutto il dominio: sono la matematica della pressione, e il cuore dell'ultimo capitolo.

14.1 Esercitazione 5 Convezione lineare in due dimensioni

$$\frac{\partial u}{\partial t} + c\,\frac{\partial u}{\partial x} + c\,\frac{\partial u}{\partial y} = 0$$

La soluzione esatta trasla il profilo iniziale lungo la diagonale: $u(x, y, t) = u_0(x - ct,\ y - ct)$. Sul piano di lavoro la soluzione è una matrice `u[i, j]` costruita con `np.meshgrid`; lo schema upwind si scrive con lo slicing su entrambi gli indici.

convezione2d.py (parte centrale)

```
x = np.linspace(0, 2, 101); dx = x[1] - x[0]
X, Y = np.meshgrid(x, x, indexing="ij")
u = np.exp(-((X-0.5)**2 + (Y-0.5)**2)/(2*0.1**2))    # campana gaussiana
dt = 0.25*dx/c                                        # CFL 2D piu' severa
for _ in range(int(round(0.5/dt))):
    u[1:,1:] = (u[1:,1:] - c*dt/dx*(u[1:,1:] - u[:-1,1:])
                         - c*dt/dx*(u[1:,1:] - u[1:,:-1]))
    u[0,:] = 0; u[:,0] = 0
```

Output

```
t = 0.50  picco 0.5768 in (1.00, 1.00)   [esatto: (1.00, 1.00)]
errore massimo 0.4232 (diffusione numerica upwind)
```

Validazione. La campana arriva esattamente in (1, 1) come impone la traslazione diagonale, ma il picco è sceso a 0,58: la diffusione numerica del primo ordine agisce ora su *due* direzioni, e su un profilo stretto (dieci celle di larghezza) il suo morso è severo. La condizione di stabilità si irrigidisce di conseguenza: per lo schema upwind 2D serve $c\,\Delta t\,(1/\Delta x + 1/\Delta y) \le 1$, da cui la scelta $\Delta t = 0{,}25\,\Delta x/c$.

14.2 Esercitazione 6 Convezione non lineare 2D: il sistema accoppiato

Nel piano il trasporto non lineare coinvolge *due* componenti di velocità che si trasportano a vicenda:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = 0, \qquad \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = 0$$

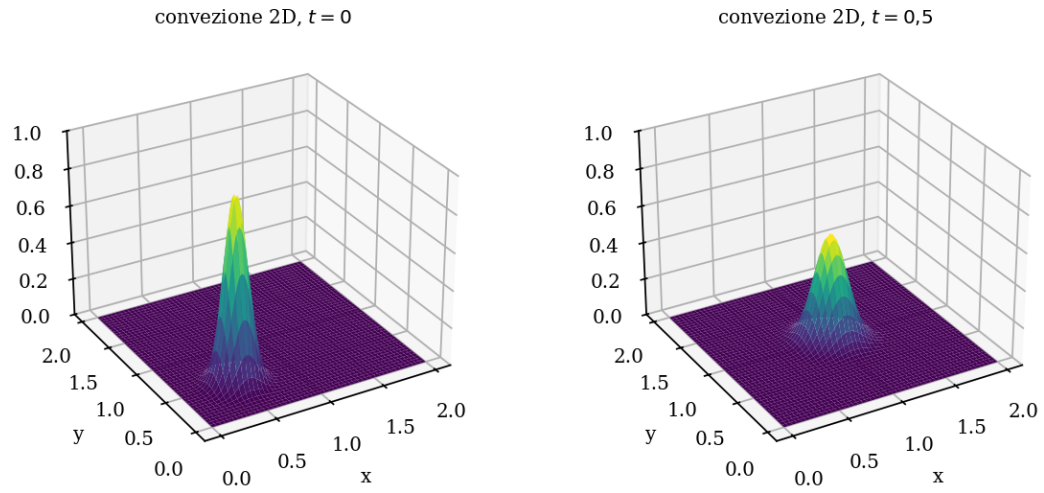


Figura 14.1. La campana gaussiana all'istante iniziale e dopo mezzo secondo di trasporto diagonale: la posizione è esatta, l'ampiezza paga il primo ordine.

È il primo *sistema* del percorso, e impone una disciplina che in Navier–Stokes sarà vitale: entrambe le equazioni vanno avanzate usando i valori *al passo precedente* (le copie `un`, `vn`), altrimenti la seconda equazione userebbe una u già aggiornata, rompendo la simultaneità del sistema.

Con condizioni iniziali identiche, $u_0 = v_0$, simmetriche nello scambio $x \leftrightarrow y$, il sistema ammette una proprietà esatta: $u(x, y, t) = v(y, x, t)$ per ogni tempo. In NumPy: `u` deve coincidere con `v.T`. È un collaudo *strutturale* che non richiede alcuna soluzione analitica.

```
Output
```

```
t = 0.25   simmetria: max|u(i,j) - v(j,i)| = 2.66e-15
picco u: 1.5710 (l'onda si irripidisce sul fronte diagonale)
```

Validazione. La simmetria è preservata fino alla precisione di macchina (10^{-15}): le due equazioni sono state discretizzate in modo davvero speculare. Se aveste scambiato un solo indice — l'errore più comune nei codici 2D — questo numero sarebbe *immediatamente* salito di quindici ordini di grandezza. Un collaudo gratuito che smaschera l'errore più insidioso della categoria: tenetelo nel vostro repertorio.

14.3 Esercitazione 7 Diffusione in due dimensioni

$$\frac{\partial u}{\partial t} = \nu\, r^2 u = \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

La campana gaussiana resta la sonda perfetta: in due dimensioni il picco decade come $s_0^2/s^2(t)$ con $s^2(t) = s_0^2 + 2\nu t$, dunque al tempo del raddoppio di varianza deve valere esattamente **1/2** (non $1/\sqrt{2}$ come in 1D: la campana si distende su due direzioni). La stabilità richiede ora $r = \nu\Delta t/\Delta x^2 \leq 1/4$ (a passi uguali nelle due direzioni).

```
Output
```

```
r = 0.125 (2r < 0.5: stabile)   t = 0.10
picco numerico 0.5000   esatto 0.5000 = 1/2
```

Validazione. Quattro cifre esatte, di nuovo: il Laplaciano discreto a cinque punti è del secondo ordine e sul profilo liscio non lascia traccia misurabile. La coppia di risultati $1/\sqrt{2}$ (1D) e $1/2$ (2D) è anche un piccolo teorema di fisica verificato al calcolatore: la diffusione è tanto più efficace quante più direzioni ha a disposizione.

14.4 Esercitazione 8 Burgers in due dimensioni

Il gran finale delle equazioni modello: convezione non lineare accoppiata *e* diffusione, per entrambe le componenti,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\, r^2 u$$

(e la gemella per v). Strutturalmente è già Navier–Stokes senza pressione: chi scrive e collauda questo codice ha scritto l'ossatura del solutore finale. Il collaudo resta quello della simmetria, che ora deve sopravvivere alla somma di *quattro* termini discretizzati per equazione:

```
Output
```

```
t = 0.25   simmetria: max|u - v^T| = 6.66e-16
picco 1.4063 (convezione + diffusione insieme)
```

Validazione. Simmetria alla precisione di macchina anche col termine viscoso; e il picco (1,41) è *minore* di quello della convezione pura dell'esercitazione 14.2 (1,57): la viscosità sta smussando l'irripidimento, esattamente il bilancio che il fronte di Burgers 1D esibiva in forma pura.

Prova tu

Riducete ν di un fattore 4 e rieseguite senza toccare Δt: o il fronte diventa troppo ripido per la griglia (oscillazioni localizzate) o va tutto bene — quale delle due dipende dal nuovo rapporto fra spessore del fronte $\sim \nu$ e passo Δx. Stimatelo *prima* di eseguire, poi verificate: è il ragionamento di risoluzione che in CFD si fa prima di ogni calcolo serio.

14.5 Esercitazione 9 L'equazione di Laplace

Cambiamo famiglia. L'equazione di Laplace,

$$r^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0$$

non contiene il tempo: descrive uno stato di *equilibrio* in cui ogni punto vale la media dei vicini, interamente determinato dalle condizioni al contorno. In aeronautica la incontrerete come equazione del flusso potenziale attorno ai profili; qui è il prototipo della matematica della pressione.

Formula chiave

Il Laplaciano discreto a cinque punti, azzerato, dà lo schema iterativo di Jacobi:

$$p_{i,j}^{k+1} = \frac{p_{i+1,j}^{k} + p_{i-1,j}^{k} + p_{i,j+1}^{k} + p_{i,j-1}^{k}}{4}$$

ogni punto diventa la media dei quattro vicini, e si ripete finché la soluzione smette di cambiare (qui: variazione massima $< 10^{-6}$). L'indice k non è il tempo fisico: è un tempo *fittizio* di rilassamento verso l'equilibrio.

Il problema di collaudo: piastra quadrata unitaria con tre lati a $p = 0$ e il lato superiore a $p = \sin(\pi x)$. La separazione delle variabili — fatela, è un esercizio d'esame universitario classico — dà la soluzione esatta in forma chiusa:

$$p(x, y) = \sin(\pi x)\,\frac{\sinh(\pi y)}{\sinh(\pi)}$$

Prompt pronto all'uso

Scrivi un programma Python con NumPy per l'equazione di Laplace sul quadrato unitario. Modello: iterazione di Jacobi vettorializzata (niente cicli sui punti), criterio d'arresto sulla variazione massima $< 10^{-6}$*. Dati: griglia* 51×51*; condizioni al contorno* $p = 0$ *su tre lati e* $p = \sin(\pi x)$ *sul lato* $y = 1$*. Formato: numero di iterazioni, valore in* $(0{,}5,\ 0{,}5)$*. Controllo: confronta con la soluzione esatta* $\sin(\pi x)\sinh(\pi y)/\sinh(\pi)$ *e stampa l'errore massimo, che deve risultare dell'ordine di* 10^{-4} *(errore di discretizzazione del secondo ordine, non dell'iterazione).*

laplace2d.py

```python
# laplace2d.py -- piastra con lato sinusoidale: Jacobi vs esatta
import numpy as np

n = 51
x = np.linspace(0, 1, n)
X, Y = np.meshgrid(x, x, indexing="ij")

p = np.zeros((n, n))
p[:, -1] = np.sin(np.pi*x)              # lato y = 1; gli altri restano 0

it = 0
while True:
    pn = p.copy()
    p[1:-1,1:-1] = 0.25*(pn[2:,1:-1] + pn[:-2,1:-1]
                         + pn[1:-1,2:] + pn[1:-1,:-2])
    it += 1
    if np.abs(p - pn).max() < 1e-6:
        break

p_es = np.sin(np.pi*X)*np.sinh(np.pi*Y)/np.sinh(np.pi)
print(f"iterazioni di Jacobi: {it}")
print(f"p(0.5, 0.5) = {p[n//2, n//2]:.5f}   esatto {p_es[n//2, n//2]:.5f}")
print(f"errore massimo = {np.abs(p - p_es).max():.2e}")
```

Output

```
iterazioni di Jacobi: 3263
p(0.5, 0.5) = 0.19886   esatto 0.19927
errore massimo = 4.14e-04
```

Validazione. Errore $4 \cdot 10^{-4}$ contro la serie esatta: è l'errore di *discretizzazione* della griglia 51×51, non quello dell'iterazione (spinta molto più giù, a 10^{-6}). Tenere distinte le due fonti di errore — quanto è fine la griglia, quanto è convergente l'iterazione — è una delle abitudini mentali che distinguono chi *usa* un solutore da chi lo *capisce*. Si noti anche il costo: 3263 iterazioni per una griglia modesta. Jacobi è il metodo più semplice e più lento della sua famiglia; Gauss–Seidel, il sovrarilassamento e il multigriglia, che studierete all'università, nascono esattamente da questa impazienza.

14.6 Esercitazione 10 L'equazione di Poisson e la soluzione manifatturata

Aggiungendo un termine di sorgente, Laplace diventa Poisson:

$$r^2 p = b(x,y) \qquad \Longrightarrow \qquad p_{i,j}^{k+1} = \frac{p_{i+1,j}^{k} + p_{i-1,j}^{k} + p_{i,j+1}^{k} + p_{i,j-1}^{k} - \Delta x^2\, b_{i,j}}{4}$$

Nel prossimo capitolo b conterrà le velocità e p sarà la pressione. Qui la usiamo per imparare la tecnica di verifica più elegante della CFD: la **soluzione manifatturata**. Il gioco è capovolto: si *sceglie* prima la soluzione, per esempio $p_{es} = \sin(\pi x)\sin(\pi y)$, la si deriva a mano per costruire la sorgente che la genera,

$$b = r^2 p_{es} = -2\pi^2 \sin(\pi x)\sin(\pi y),$$

e si dà al codice quella sorgente: se il codice è corretto, deve restituire la soluzione scelta. E poiché la soluzione esatta è nota *per costruzione*, si può misurare l'ordine di convergenza:

poisson2d.py (parte centrale)

```
X, Y = np.meshgrid(x, x, indexing="ij")
b = -2*np.pi**2*np.sin(np.pi*X)*np.sin(np.pi*Y)    # sorgente costruita
p = np.zeros((n, n))                               # bordo: p = 0 esatto
for _ in range(20000):
    pn = p.copy()
    p[1:-1,1:-1] = 0.25*(pn[2:,1:-1] + pn[:-2,1:-1] + pn[1:-1,2:]
                         + pn[1:-1,:-2] - dx**2*b[1:-1,1:-1])
    if np.abs(p - pn).max() < 1e-8:
        break
errore = np.abs(p - np.sin(np.pi*X)*np.sin(np.pi*Y)).max()
```

Output

```
n = 26   dx = 0.0400   errore massimo = 0.00131
n = 51   dx = 0.0200   errore massimo = 0.00032
rapporto degli errori = 4.04   (atteso 4: schema del secondo ordine)
```

Validazione. Dimezzando il passo l'errore si divide per 4,04: lo schema mantiene la promessa del secondo ordine con uno scarto dell'uno per cento. Questa procedura — soluzione manifatturata più studio di convergenza — non è un esercizio scolastico: è il protocollo standard con cui si *verifica* un codice CFD professionale prima di fidarsene, e saperla eseguire vi metterà un passo avanti in qualunque laboratorio universitario.

Prova tu

Sostituite Jacobi con Gauss–Seidel: basta aggiornare `p` «in posto», usando i valori già nuovi appena disponibili (il ciclo diventa sui punti, oppure si usa l'aggiornamento a scacchiera per restare vettorializzati). Contate le iterazioni: dovrebbero circa dimezzarsi a parità di tolleranza. Verificate che l'errore finale contro la soluzione manifatturata resti identico: il *metodo* iterativo cambia la velocità, non la soluzione discreta a cui si converge.

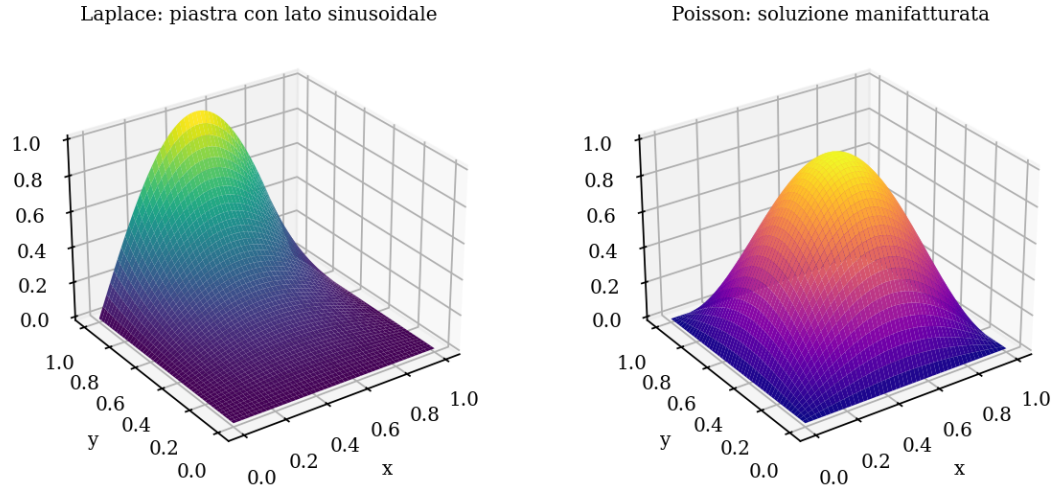


Figura 14.2. Le due equazioni ellittiche risolte per iterazione: a sinistra Laplace (la piastra col lato sinusoidale), a destra Poisson con la sorgente manifatturata.

Capitolo 15

Le equazioni di Navier–Stokes al calcolatore

Nota

Il traguardo. Tutti i mattoni sono collaudati: convezione non lineare accoppiata (esercitazioni 14.2 e 14.4), diffusione (14.3), solutore di Poisson (14.6). Resta da capire il ruolo della pressione, che è la vera idea nuova di questo capitolo; poi i mattoni si montano e si risolvono due problemi canonici della disciplina: la cavità con coperchio mobile e il canale piano.

15.1 Il problema della pressione

Riprendiamo il sistema incomprimibile del capitolo 12. Le due equazioni di quantità di moto dicono come evolvono u e v; ma la pressione non ha un'equazione di evoluzione propria: compare solo attraverso il suo gradiente, e la continuità $r \cdot \mathbf{V} = 0$ non la contiene affatto. La pressione, in un flusso incomprimibile, non è una variabile che evolve: è un *vincolo* — il moltiplicatore che istante per istante assume esattamente il valore necessario a mantenere il campo di velocità a divergenza nulla. È la stessa logica della reazione vincolare in statica: non la si assegna, la si *ricava* dall'equilibrio.

Come ricavarla? Prendendo la divergenza dell'equazione di quantità di moto e imponendo la continuità, i termini instazionario e viscoso si annullano e resta un'equazione di **Poisson per la pressione**:

Formula chiave

$$r^2 p = \rho \left[\frac{1}{\Delta t} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) - \left(\frac{\partial u}{\partial x} \right)^2 - 2 \frac{\partial u}{\partial y} \frac{\partial v}{\partial x} - \left(\frac{\partial v}{\partial y} \right)^2 \right]$$

Il primo termine fra parentesi merita un commento: in teoria la divergenza è nulla e sparirebbe; nella pratica discreta la divergenza del passo precedente non è mai esattamente zero, e tenerla nella sorgente — divisa per Δt — agisce da correttore che a ogni passo *riassorbe* l'errore di continuità accumulato, invece di lasciarlo crescere.

L'algoritmo di ogni passo temporale è dunque un montaggio di pezzi noti:

1. con u, v correnti si costruisce la sorgente b;
2. si risolve $r^2 p = b$ con il solutore di Poisson dell'esercitazione 14.6 (qui: un numero fissato di iterazioni interne);
3. si avanzano u e v con convezione, gradiente di pressione e diffusione — lo schema di Burgers 2D più il termine $-\frac{\Delta t}{\rho} r p$;
4. si impongono le condizioni al contorno.

15.2 Esercitazione 11 La cavità con coperchio mobile

Il banco di prova per antonomasia dei solutori incomprimibili: una scatola quadrata di lato L piena di fluido fermo, il cui coperchio inizia a scorrere a velocità costante U. Il coperchio trascina il fluido, che non potendo uscire si avvolge in un vortice: geometria elementare, fisica completa (strato limite, ricircolo, angoli singolari),

e per questo il caso su cui da mezzo secolo ogni nuovo codice viene messo alla prova. Il numero di Reynolds $Re = UL/\nu$ governa tutto: qui $U = 1$, $L = 1$, $\nu = 0{,}05$, dunque $Re = 20$, un regime dolcemente viscoso adatto alla nostra griglia 41×41.

Le condizioni al contorno: aderenza su tutte le pareti ($u = v = 0$), tranne il coperchio dove $u = U$, $v = 0$; per la pressione, derivata normale nulla sulle pareti e valore di riferimento fissato sul coperchio.

Prompt pronto all'uso

Scrivi un programma Python con NumPy per la cavità con coperchio mobile. Modello: Navier–Stokes incomprimibile su griglia 41×41; a ogni passo (1) sorgente b dalle derivate centrate delle velocità, (2) 60 iterazioni di Jacobi per la Poisson della pressione con Neumann sulle pareti, (3) avanzamento di u e v con upwind sulla convezione, centrate su pressione e diffusione, usando le copie del passo precedente. Dati: $\rho = 1$, $\nu = 0{,}05$, $U = 1$, $\Delta t = 10^{-3}$, 1500 passi. Formato: u e v al centro, massimo e media di $|r \cdot \mathbf{V}|$. Controlli: u sul coperchio deve restare esattamente 1; la divergenza nell'interno deve risultare piccola rispetto a $U/L = 1$; al centro u deve essere negativa (il ricircolo di ritorno).

`cavita.py (struttura del passo temporale)`

```
for passo in range(nt):
    un, vn = u.copy(), v.copy()

    b = sorgente(un, vn)                  # 1) termine noto della Poisson
    for _ in range(nit):                  # 2) pressione: 60 iter. Jacobi
        pn = p.copy()
        p[1:-1,1:-1] = (0.25*(pn[2:,1:-1] + pn[:-2,1:-1]
                        + pn[1:-1,2:] + pn[1:-1,:-2])
                        - dx**2/4*b[1:-1,1:-1])
        p[0,:] = p[1,:];  p[-1,:] = p[-2,:]     # Neumann sui fianchi
        p[:,0] = p[:,1];  p[:,-1] = 0           # riferimento sul coperchio

    u[1:-1,1:-1] = (un[1:-1,1:-1]         # 3) quantita' di moto x
        - un[1:-1,1:-1]*dt/dx*(un[1:-1,1:-1] - un[:-2,1:-1])
        - vn[1:-1,1:-1]*dt/dy*(un[1:-1,1:-1] - un[1:-1,:-2])
        - dt/(2*rho*dx)*(p[2:,1:-1] - p[:-2,1:-1])
        + nu*dt/dx**2*(un[2:,1:-1] - 2*un[1:-1,1:-1] + un[:-2,1:-1])
        + nu*dt/dy**2*(un[1:-1,2:] - 2*un[1:-1,1:-1] + un[1:-1,:-2]))
    # ... equazione gemella per v ...

    u[:,0] = 0; u[0,:] = 0; u[-1,:] = 0   # 4) aderenza alle pareti
    u[:,-1] = U                           #    coperchio: u = 1
    v[:,0] = 0; v[0,:] = 0; v[-1,:] = 0; v[:,-1] = 0
```

Output

```
Re = 20   t = 1.50
u al centro = -0.1655   v al centro = +0.0107
u max = 1.000 (sul coperchio: 1.000)   u min = -0.1667
div: max globale 6.70 (spigoli del coperchio)
div interno (lontano dal coperchio): max 0.1248   medio 0.00184
```

Validazione. Tre controlli, tutti superati. Primo: il coperchio conserva esattamente $u = 1$ e le pareti $u = v = 0$ — le condizioni al contorno reggono. Secondo: al centro $u = -0{,}17$, *negativa*: il fluido di ritorno del vortice scorre in verso opposto al coperchio, come la fisica del ricircolo impone. Terzo, il più importante: la divergenza media nell'interno è $2 \cdot 10^{-3}$, duecento volte più piccola della scala U/L — la Poisson sta facendo

il suo mestiere di vincolo. Il massimo globale (6,7) vive tutto nei due spigoli superiori, dove la condizione al contorno è *matematicamente* discontinua (u salta da 0 a 1 nel punto d'angolo): è la nota singolarità degli spigoli del coperchio, presente in ogni soluzione di questo problema, e riconoscerla come tale — invece di nasconderla in una media globale — fa parte dell'onestà del mestiere.

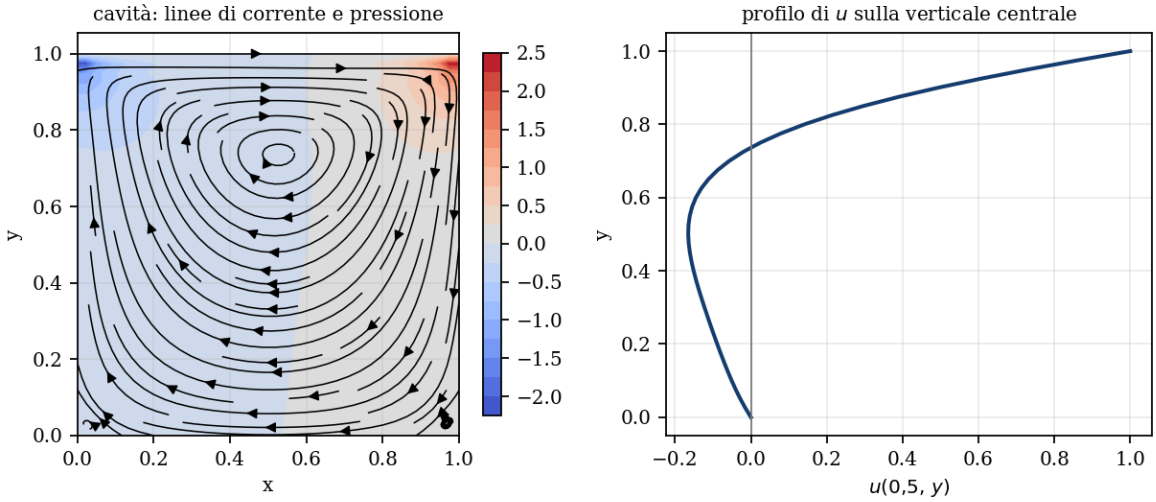


Figura 15.1. La cavità a $Re = 20$: a sinistra linee di corrente sul campo di pressione (il vortice riempie la scatola, coi minimi di pressione agli spigoli del coperchio); a destra il profilo di u sulla verticale centrale, positivo in alto e negativo nel ritorno.

Prova tu

Portate Re a 100 dimezzando due volte ν: il vortice principale si ingrandisce e il suo centro migra verso il centro della scatola; verificate che serva anche ridurre Δt (perché?). I profili di u sulla verticale centrale a $Re = 100$ su questo problema sono tabulati in letteratura da decenni come riferimento: confrontarvisi è il primo esercizio da laboratorio universitario di CFD, e il vostro codice è già in grado di tentarlo.

15.3 Esercitazione 12 Il canale piano: ritrovare Poiseuille

Chiusura ad anello. Un canale piano di altezza H con fluido spinto da una forzante costante F per unità di massa (l'equivalente di un gradiente di pressione imposto): se il moto è parallelo alle pareti e uniforme in x, le equazioni di Navier–Stokes collassano in una sola riga,

$$\frac{\partial u}{\partial t} = \frac{F}{\rho} + \nu \frac{\partial^2 u}{\partial y^2}$$

— la *diffusione con sorgente* — e il regime stazionario ha soluzione esatta: il profilo parabolico di Poiseuille,

$$u(y) = \frac{F}{2\rho\nu}\, y\,(H - y), \qquad u_{max} = \frac{F H^2}{8\rho\nu}$$

che per $F = 1, H = 2, \rho = 1, \nu = 0{,}1$ vale $u_{max} = 5$ esatto al centro. È una delle pochissime soluzioni esatte di Navier–Stokes, la stessa che regola la portata nei condotti e nei circuiti idraulici di bordo: il traguardo perfetto, perché il solutore numerico deve *ritrovarla* partendo dal fluido fermo.

canale.py

```python
# canale.py -- avviamento del canale piano fino al regime di Poiseuille
import numpy as np

n, H = 41, 2.0
y = np.linspace(0, H, n); dy = y[1] - y[0]
nu, F, rho = 0.1, 1.0, 1.0
dt = 4e-3                                   # r = nu dt/dy^2 = 0.16 < 0.5

u = np.zeros(n)                             # fluido inizialmente fermo
for passo in range(1, 12001):
    un = u.copy()
    u[1:-1] = un[1:-1] + dt*(F/rho
              + nu*(un[2:] - 2*un[1:-1] + un[:-2])/dy**2)
    u[0] = 0.0; u[-1] = 0.0                 # aderenza alle pareti
    if np.abs(u - un).max() < 1e-8:         # regime raggiunto
        break

u_es = F/(2*rho*nu)*y*(H - y)               # parabola di Poiseuille
print(f"passi eseguiti: {passo}   t = {passo*dt:.0f} s")
print(f"u max numerico {u.max():.4f}   esatto F H^2/(8 rho nu) = "
      f"{F*H**2/(8*rho*nu):.4f}")
print(f"errore massimo sul profilo = {np.abs(u - u_es).max():.2e}")
```

Output

```
passi eseguiti: 12000   t = 48 s
u max numerico 5.0000   esatto F H^2/(8 rho nu) = 5.0000
errore massimo sul profilo = 3.71e-05
```

Validazione. Il solutore, partito dal fluido fermo, converge alla parabola esatta con un errore di $4 \cdot 10^{-5}$ su un massimo di 5: quattro cifre significative. E c'è un bonus fisico nel tempo di avviamento: il transitorio dura circa $H^2/\nu = 40$ secondi, il tempo che la viscosità impiega a *diffondere* l'informazione delle pareti fino al centro del canale — lo stesso meccanismo, e la stessa scala dei tempi, dell'accrescimento di uno strato limite.

Osservazione

Guardate indietro: il percorso è partito da $u_i^{n+1} = u_i^n - \sigma(u_i^n - u_{i-1}^n)$, una riga, ed è arrivato a un solutore di Navier–Stokes che riproduce benchmark e soluzioni esatte. Nessun passaggio ha richiesto strumenti che non possedeste già dalla Parte I: solo l'accumulo paziente di mattoni *validati*. È il metodo, non il talento, che porta fin qui.

Verso l'università

Questo percorso vi ha dato le fondamenta concettuali: discretizzare, stimare la stabilità, verificare l'ordine di convergenza, validare contro soluzioni esatte e benchmark. Che cosa vi aspetta dopo, e con quali parole chiave cercarlo: i *volumi finiti*, la discretizzazione conservativa usata dai codici industriali al posto delle differenze finite; le griglie non strutturate, per geometrie reali; gli schemi impliciti e di ordine alto, che superano i limiti di Δt visti qui; il grande capitolo della *turbolenza* (RANS, i modelli di chiusura, LES), che domina ogni flusso aeronautico reale a numeri di Reynolds di milioni; e gli strumenti aperti con cui tutto questo si pratica, dai solutori generalisti a quelli nati per l'aerodinamica esterna, che potrete installare e usare da subito.

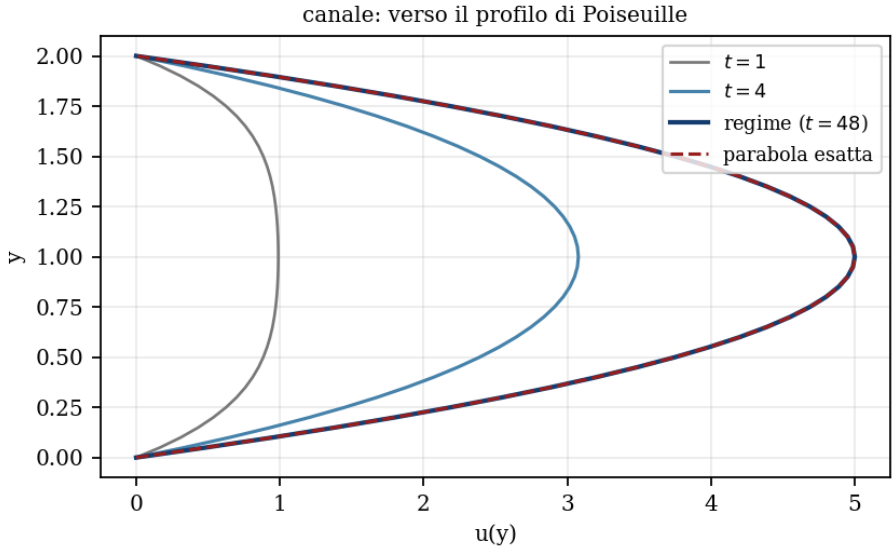


Figura 15.2. L'avviamento del canale: i profili a $t = 1$ e $t = 4$ ancora in crescita, e il regime che si adagia sulla parabola esatta di Poiseuille (tratteggio).

Un'ultima avvertenza, che è poi la tesi dell'intero volume: quei codici professionali produrranno immagini bellissime *qualunque* input diate loro. Distinguere un risultato convergente da uno colorato, un errore di discretizzazione da uno di modello, sarà compito vostro — e chi ha verificato di persona un rapporto di convergenza pari a 4,04 non lo dimentica più.

Dialoga con l'IA

Ultimo dialogo del volume. Consegnate all'assistente il vostro `cavita.py` e chiedete: *"Trasforma le iterazioni interne di Jacobi in Gauss–Seidel e aggiungi un criterio d'arresto sulla divergenza massima invece che sul numero fisso di iterazioni"*. Poi collaudate come ormai sapete: la divergenza interna deve scendere, il campo di velocità non deve cambiare in modo apprezzabile, il coperchio deve restare a $u = 1$. Se tutte e tre le cose accadono, avete appena fatto — in piccolo, ma per intero — il lavoro dell'ingegnere numerico: migliorare un solutore *dimostrando* di non averlo rotto.

Capitolo A

Prontuario essenziale

Nota

Questo prontuario è pensato per stare aperto *accanto* al calcolatore mentre si programma: non si legge, si consulta. È diviso in tre blocchi: il **linguaggio**, i **dati** del corso e il **metodo** di lavoro. L'ultimo paragrafo — la lista di controllo prima della consegna — è quello che conviene rileggere sempre.

A.1 Eseguire un programma

Operazione	Comando
Eseguire uno script	`python3 nome.py`
Sessione interattiva	`python3` (si esce con `exit()`)
Installare una libreria	`pip install numpy matplotlib`
Interrompere l'esecuzione	`Ctrl+C`

Il file deve avere estensione `.py` ed essere salvato nella cartella da cui si lancia il comando. Se compare `ModuleNotFoundError`, la libreria non è installata; se compare `FileNotFoundError`, siete nella cartella sbagliata.

A.2 Tipi e operatori

Tipo	Esempio	Nota
`int`	`n = 6`	intero, precisione illimitata
`float`	`rho = 1.225`	decimale, circa 16 cifre significative
`bool`	`ok = True`	vale `True` oppure `False`
`str`	`s = "NACA 2412"`	testo, fra apici singoli o doppi
`list`	`q = [0, 500]`	modificabile, indice da 0
`tuple`	`p = (2.0, 3.0)`	non modificabile
`dict`	`d = {"b": 10}`	coppie chiave–valore

Operatore	Significato	Esempio	Risultato
`+ - * /`	operazioni usuali	`7/2`	`3.5`
`**`	elevamento a potenza	`V**2`	quadrato
`//`	divisione intera	`7//2`	`3`
`%`	resto	`7%2`	`1`
`== !=`	uguale, diverso	`n == 6`	`True`
`< <= > >=`	confronti	`M < 0.3`	`True/False`
`and or not`	operatori logici	`h >= 0 and h <= 11000`	
`+= -=`	aggiorna in luogo	`s += V*dt`	

Osservazione

La divisione `/` restituisce *sempre* un `float`, anche fra interi: `4/2` vale `2.0`. È il contrario di quanto accade in C, e vi risparmia la classe di errori più insidiosa: il troncamento silenzioso.

A.3 Sintassi di base

Costrutto	Sintassi
Assegnazione	`rho = 1.225`
Assegnazione multipla	`T, p, rho = isa(h)`
Commento	`# densita' [kg/m^3]`
Input numerico	`V = float(input("V [m/s]:  "))`
Output formattato	`print(f"q = {q:.1f} Pa")`
Decisione	`if` / `elif` / `else` + indentazione
Ciclo su intervallo	`for h in range(0, 12001, 1000):`
Ciclo su lista	`for c in corde:`
Ciclo condizionale	`while V < VR:`
Uscita anticipata	`break` (esce), `continue` (salta al giro dopo)
Funzione	`def isa(h):  ...  return T, p, rho`
Segnalare un errore	`raise ValueError("quota fuori dominio")`
Modulo matematico	`import math`
NumPy	`import numpy as np`
Matplotlib	`import matplotlib.pyplot as plt`
Uso di un proprio file	`from atmosfera20 import isa`

Osservazione

L'indentazione (quattro spazi) non è estetica: *è* la sintassi. In Python il blocco di un `if` o di un `for` è definito da come le righe sono rientrate. Mescolare spazi e tabulazioni produce `IndentationError` o, peggio, un programma che gira eseguendo un blocco diverso da quello voluto.

Le tre forme di `range`

Forma	Genera	Nota
`range(5)`	`0 1 2 3 4`	parte da 0
`range(2, 6)`	`2 3 4 5`	estremo destro escluso
`range(0, 12001, 1000)`	`0 1000 ... 12000`	il passo è il terzo argomento

L'estremo destro è *sempre* escluso: per arrivare a 12 000 m compresi occorre scrivere `12001`. È la causa numero uno delle tabelle atmosferiche che si fermano una riga prima.

A.4 Stampa formattata

La *f-string* inserisce il valore di un'espressione fra parentesi graffe e ne governa il formato dopo i due punti, secondo lo schema

```
{valore:[allineamento][segno][larghezza].[decimali][tipo]}
```

dove `tipo` vale `f` (decimale), `e` (scientifica), `d` (intero), `s` (testo), `%` (percentuale).

Formato	Risultato	Uso tipico
`{x:.2f}`	`1234.57`	valore con due decimali
`{x:10.2f}`	`|   1234.57|`	incolonnare una tabella
`{x:<10.2f}`	`|1234.57   |`	allineare a sinistra
`{x:+.1f}`	`+1234.6`	mostrare sempre il segno
`{x:.3e}`	`1.235e+03`	numeri molto grandi o piccoli
`{x:.0f}`	`1235`	arrotondare all'unità
`{n:5d}`	`|   42|`	interi incolonnati
`{n:05d}`	`00042`	numerazione con zeri
`{s:>8s}`	`|     ala|`	testo allineato a destra
`{r:.1%}`	`85.0%`	rendimenti e rapporti

lo schema di tabella da riusare sempre

```
print("  h[m]      T[K]")
for h, T in [(0, 288.15), (5000, 255.65), (11000, 216.65)]:
    print(f"  {h:6d} {T:8.2f}")
```

Output

```
h[m]      T[K]
       0   288.15
    5000   255.65
   11000   216.65
```

Osservazione

La larghezza si sceglie *una volta* guardando il valore più lungo della colonna, e si mantiene identica nell'intestazione e nel ciclo. Una tabella disallineata non è un difetto estetico: rende impossibile il controllo a colpo d'occhio, che è il primo strumento di validazione di cui disponete.

A.5 Contenitori: liste, tuple, dizionari

Operazione	Sintassi
Creare	`v = [1.0, 2.0, 3.0]`
Primo, ultimo	`v[0]`, `v[-1]`
Sottolista	`v[1:3]` (secondo e terzo elemento)
Lunghezza	`len(v)`
Aggiungere in coda	`v.append(4.0)`
Ordinare	`v.sort()` oppure `sorted(v)`
Minimo, massimo, somma	`min(v)`, `max(v)`, `sum(v)`
Scorrere con indice	`for i, x in enumerate(v):`
Scorrere in parallelo	`for h, T in zip(quote, temperature):`
Costruire da formula	`q = [0.5*1.225*V**2 for V in velocita]`
Dizionario	`ala = {"b":  10.0, "S":  13.5}`
Leggere per chiave	`ala["S"]`
Scorrere un dizionario	`for chiave, valore in ala.items():`

La penultima riga della prima metà — la *comprensione di lista* — costruisce una lista applicando una formula a ogni elemento di un'altra, in una riga sola: è il ponte concettuale verso gli array NumPy.

A.6 Funzioni e moduli propri

la forma canonica di una funzione

```
def quota_isa(h):
    """Restituisce (T, p, rho) alla quota h in metri.

    Valida per 0 <= h <= 11000 m (troposfera ISA).
    """
    if h < 0 or h > 11000:
        raise ValueError(f"quota {h} m fuori dalla troposfera")
    T = 288.15 - 0.0065*h
    p = 101325.0*(T/288.15)**5.25593
    return T, p, p/(287.05*T)
```

Elemento	Regola
Nome	minuscolo, descrittivo, con _ fra le parole
Docstring	fra tripli apici, subito sotto il `def`
Dominio	controllato all'inizio, con `raise ValueError`
Valori multipli	`return T, p, rho`
Argomento con difetto	`def q(V, rho=1.225):`
Riuso in altri file	`from atmosfera20 import isa`

Osservazione

Un file `.py` che contiene solo definizioni di funzioni è un *modulo*: si importa da altri programmi e diventa parte del vostro strumentario. Perché l'importazione funzioni, i due file devono stare nella stessa cartella. È così che nascono `atmosfera20.py` e `vettori2d.py`: non da un esercizio in più, ma dalla decisione di non riscrivere mai ciò che è già collaudato.

A.7 Il modulo `math`

Funzione	Significato	Nota
`math.sqrt(x)`	radice quadrata	errore se `x<0`
`math.exp(x)`	e^x	atmosfera isoterma
`math.log(x)`	logaritmo naturale	formula di Breguet
`math.sin cos tan`	funzioni goniometriche	**in radianti**
`math.asin acos atan`	funzioni inverse	restituiscono radianti
`math.atan2(y, x)`	arcotangente a due argomenti	quadrante corretto
`math.radians(g)`	gradi $\to$ radianti	prima del calcolo
`math.degrees(r)`	radianti $\to$ gradi	solo in stampa
`math.hypot(x, y)`	$\sqrt{x^2+y^2}$	moduli di vettori
`math.ceil(x)`	arrotonda per eccesso	numero di rivetti
`math.floor(x)`	arrotonda per difetto	
`math.pi`, `math.e`	costanti	
`math.inf`	infinito	confronti iniziali

Osservazione

Le tre da tenere sempre a mente: `atan2` (mai `atan` su un vettore, perché perde il quadrante), la coppia `radians`/`degrees` (Python lavora in radianti) e `ceil` (il numero di rivetti si arrotonda *sempre* per eccesso:

con uno in meno la verifica non passa).

A.8 NumPy in una pagina

Operazione	Sintassi
Da lista	`v = np.array([1.0, 2.0, 3.0])`
Intervallo con passo	`np.arange(0, 12001, 1000)`
Intervallo con *n* punti	`np.linspace(0, 11000, 500)`
Vettori nulli, unitari	`np.zeros(n)`, `np.ones(n)`
Operazioni su tutto	`CD = CD0 + k*CL**2`
Funzioni vettorializzate	`np.sqrt`, `np.exp`, `np.sin`, `np.log`
Gradi e radianti	`np.radians(a)`, `np.degrees(a)`
Minimo, massimo, media	`v.min()`, `v.max()`, `v.mean()`
Dove sta il minimo	`i = v.argmin()` e poi `V[i]`
Somma, prodotto scalare	`v.sum()`, `np.dot(a, b)`
Modulo di un vettore	`np.linalg.norm(v)`
Prodotto vettoriale	`np.cross(a, b)`
Integrale numerico	`np.trapezoid(y, x)`
Selezione condizionata	`v[v > 0.5]`
Maschera booleana	`m = (V >= 30) & (V <= 60)`
Limitare a un intervallo	`np.clip(x, -1.0, 1.0)`
Arrotondare per stampa	`np.round(v, 3)`

Osservazione

`argmin` e `argmax` restituiscono l'*indice*, non il valore: servono per rispondere alla domanda «a quale velocità?» anziché «quanto vale?». È la coppia `Pn.min()` e `V[Pn.argmin()]` che ha dato la velocità di potenza minima nella scheda 10.2.
Attenzione a `np.trapezoid`: nelle versioni di NumPy precedenti alla 2.0 la funzione si chiamava `np.trapz`. Se l'ambiente segnala che il nome non esiste, provate l'altro.

A.9 Matplotlib: ricettario dei grafici tecnici

Obiettivo	Comando
Figura e assi	`fig, ax = plt.subplots(figsize=(6, 4))`
Curva	`ax.plot(x, y, label="Pn")`
Punti	`ax.scatter(x, y)`
Frecce (vettori)	`ax.quiver(x0, y0, dx, dy, angles="xy", scale_units="xy", scale=1)`
Scala semilogaritmica	`ax.semilogy(h, p)`
Riempimento fra curve	`ax.fill_between(x, y1, y2, alpha=0.3)`
Retta orizzontale	`ax.axhline(3.8, ls="-", color="red")`
Etichette e titolo	`ax.set_xlabel(...)`, `ax.set_title(...)`
Griglia	`ax.grid(alpha=0.3)`
Legenda	`ax.legend()`
Limiti degli assi	`ax.set_xlim(0, 80)`
Scala uguale	`ax.set_aspect("equal")`
Annotazione con freccia	`ax.annotate("Emax", xy=..., xytext=...)`
Due grafici affiancati	`fig, (a1, a2) = plt.subplots(1, 2)`
Salvare	`plt.savefig("polare.png", dpi=150, bbox_inches="tight")`
Mostrare a schermo	`plt.show()`

Osservazione

`ax.set_aspect("equal")` è obbligatorio ogni volta che il disegno ha significato geometrico: profili alari, vettori, piante alari. Senza di esso gli assi hanno scale diverse, un angolo di 45° appare di 70° e un profilo appare più spesso di quanto sia. Per i diagrammi funzionali (P_n contro V, la polare, i diagrammi di sollecitazione) non serve, anzi peggiora la leggibilità.

A.10 Costanti del corso

Simbolo	Valore	Unità	Significato
T_0	288,15	K	temperatura ISA al mare
p_0	101 325	Pa	pressione ISA al mare
ρ_0	1,225	kg/m^3	densità ISA al mare
a_0	340,29	m/s	velocità del suono al mare
λ	$6{,}5 \times 10^{-3}$	K/m	gradiente troposferico
g	9,806 65	m/s^2	gravità standard
R	287,05	J/(kg K)	costante dell'aria secca
γ	1,4	–	rapporto dei calori specifici
T_s	216,65	K	temperatura in bassa stratosfera
h_{trop}	11 000	m	quota della tropopausa
p_{11}	22 632	Pa	pressione alla tropopausa
μ_0	$1{,}7894 \times 10^{-5}$	Pa s	viscosità dinamica al mare

Due costanti *derivate* che conviene conoscere a memoria, perché compaiono in quasi ogni calcolo atmosferico:

$$\frac{g}{R\lambda} = 5{,}25593, \qquad \frac{T_0}{\lambda} = 44\,330{,}8\ \text{m}$$

La prima è l'esponente della formula barometrica troposferica; la seconda è la quota alla quale la temperatura ISA si annullerebbe estrapolando il gradiente, e consente la forma compatta $T/T_0 = 1 - h/44330{,}8$.

A.11 L'atmosfera ISA ai livelli di volo

FL	h [m]	T [K]	t [°C]	p [Pa]	σ	a [m/s]
0	0,0	288,15	15,00	101325	1,0000	340,29
50	1524,0	278,24	5,09	84307	0,8617	334,39
100	3048,0	268,34	−4,81	69681	0,7385	328,39
150	4572,0	258,43	−14,72	57182	0,6292	322,27
200	6096,0	248,53	−24,62	46563	0,5328	316,03
250	7620,0	238,62	−34,53	37601	0,4481	309,67
300	9144,0	228,71	−44,44	30089	0,3741	303,17
350	10668,0	218,81	−54,34	23842	0,3099	296,53
400	12192,0	216,65	−56,50	18754	0,2462	295,07
450	13716,0	216,65	−56,50	14747	0,1936	295,07

Valori generati con `atmosfera20.py` (scheda 9.3): servono come *riferimento di collaudo* per i vostri programmi. Dal FL400 in poi la temperatura è costante — siamo in stratosfera — e infatti la velocità del suono smette di calare.

Osservazione

Il livello di volo è la quota di pressione in centinaia di piedi: FL350 significa 35 000 ft, cioè 10 668 m. La colonna $\sigma = \rho/\rho_0$ è quella che serve nelle prestazioni: a FL350 vale 0,31, dunque la velocità vera è

$1/\sqrt{0{,}31} = 1{,}80$ volte quella indicata. È in questo numero che sta la ragione economica del volo in quota.

Pressione dinamica di riferimento (a ρ_0)

V [m/s]	25	50	75	100	150
q [Pa]	382,8	1531,2	3445,3	6125,0	13781,3

Poiché $q \propto V^2$, raddoppiando la velocità la pressione dinamica *quadruplica*: $1531 \to 6125$. È il controllo mentale più rapido che esista su un risultato aerodinamico.

A.12 Conversioni aeronautiche

Da	A	Fattore
1 kt	m/s	0,514444
1 kt	km/h	1,852
1 ft	m	0,3048
1 NM	m	1852
1 in	mm	25,4
1 lb	kg	0,453592
1 lbf	N	4,448222
1 kgf	N	9,80665
1 slug	kg	14,593903
1 psi	Pa	6894,757
1 ksi	MPa	6,894757
1 bar	Pa	100 000
1 inHg	Pa	3386,389
1 hp	W	745,700
1 CV	W	735,499

Le prime quattro sono *esatte* per convenzione ICAO: il nodo vale 1852/3600 m/s e il piede esattamente 0,3048 m. Da $1\,\text{kt} = 1{,}852\,\text{km/h}$ discende la regola mnemonica dei piloti: dai nodi ai chilometri orari si moltiplica per 1,85, cioè si aggiunge poco meno della metà.

Per le temperature, che non sono grandezze proporzionali:

$$T[\text{K}] = t[^\circ\text{C}] + 273{,}15, \qquad t[^\circ\text{C}] = \tfrac{5}{9}\left(t[^\circ\text{F}] - 32\right)$$

A.13 Proprietà indicative dei materiali

Materiale	E [GPa]	σ_{sn} [MPa]	σ_R [MPa]	ρ [kg/m^3]
Lega Al 2024-T3	73	345	483	2780
Lega Al 7075-T6	72	503	572	2810
Lega Al 6061-T6	69	276	310	2700
Acciaio 4130 normalizzato	205	435	670	7850
Lega Ti-6Al-4V	114	880	950	4430

Osservazione

Sono *valori indicativi di letteratura*, adatti agli esercizi e agli ordini di grandezza. Una verifica strutturale reale richiede i dati di specifica del semilavorato (MMPDS, norme EN), che dipendono da spessore, direzione di laminazione e trattamento termico. Per le leghe di alluminio vale la regola pratica

$\tau_R \simeq (0{,}5 \div 0{,}6)\,\sigma_R$, utile per una prima stima delle verifiche a taglio.
Si osservi il rapporto E/ρ: praticamente identico per tutte e tre le leghe di alluminio, e non molto diverso per acciaio e titanio. La scelta del materiale in aeronautica non si gioca dunque sulla rigidezza specifica, ma sulla resistenza specifica, sul comportamento a fatica e sulla resistenza a corrosione.

A.14 Formulario operativo: dove trovare cosa

Indice inverso delle schede: individuata la formula, la colonna di destra dice dove è implementata, eseguita e collaudata.

Grandezza	Formula	Scheda
Atmosfera e aerodinamica di base		
Temperatura ISA	$T = T_0 - \lambda h$	9.2
Pressione ISA	$p = p_0 (T/T_0)^{5{,}25593}$	9.2
Stratosfera isoterma	$p = p_{11} e^{-g(h-11000)/(RT_s)}$	9.3
Densità	$\rho = p/(RT)$	9.2
Pressione dinamica	$q = \frac{1}{2}\rho V^2$	9.7
Velocità dal Pitot	$V = \sqrt{2\Delta p/\rho}$	9.7
Velocità del suono	$a = \sqrt{\gamma R T}$	9.8
Numero di Reynolds	$Re = \rho V c/\mu$	9.8
Viscosità (Sutherland)	$\mu = 1{,}458\cdot 10^{-6} T^{1,5}/(T + 110{,}4)$	9.8
Spinta aerostatica	$F_A = \rho V g$	9.5
Geometria alare		
Superficie	$S = \frac{c_r + c_t}{2}\, b$	9.9
Allungamento	$AR = b^2/S$	9.9
Corda media aerodin.	$\bar{c} = \frac{2}{3} c_r \frac{1+\lambda_r+\lambda_r^2}{1+\lambda_r}$	9.9
Spessore NACA 4 cifre	$y_t/c = 5t(0{,}2969\sqrt{\bar{x}} - \dots)$	9.10
Meccanica del volo		
Velocità di stallo	$V_s = \sqrt{2W/(\rho S C_{Lmax})}$	10.1
Polare parabolica	$C_D = C_{D0} + k C_L^2$	10.2
Efficienza massima	$E_{max} = 1/(2\sqrt{C_{D0} k})$	10.1
Potenza necessaria	$P_n = D\, V$	10.2
Rateo di salita	$RC = (P_d - P_n)/W$	10.3
Fattore di carico	$n = 1/\cos\varphi$	10.4
Raggio di virata	$R = V^2/(g \tan\varphi)$	10.4
Planata	$\tan\gamma = 1/E,\ d = h\,E$	10.5
Corsa di decollo	$m\,a = T - D - \mu_r (W - L)$	10.6
Distanza di Breguet	$d = \frac{\eta}{g c_s} E \ln\frac{W_0}{W_1}$	10.8
Velocità di manovra	$V_A = V_s \sqrt{n_{max}}$	10.9
Raffica (Pratt)	$\Delta n = \rho_0 U_{de} V C_{L\alpha} K_g/(2W/S)$	10.10
Strutture		
Navier	$\sigma = M\,c/I$	11.1
Huygens–Steiner	$I = \sum [b_i h_i^3/12 + A_i d_i^2]$	11.2
Flusso di taglio	$q = T/d,\ \tau = q/t$	11.3
Taglio e momento	$T(y) = \int_y^s \ell\, \mathrm{d}\xi$	11.4
Bredt (prima)	$\tau = M_t/(2\Omega t)$	11.5
Bredt (seconda)	$\theta = M_t \pi_m/(4\Omega^2 G t)$	11.5
Rivetti a taglio	$n \geq F/(\tau_{amm}\, \pi d^2/4)$	11.6
Carico critico	$P_{cr} = \pi^2 EI/L^2$	11.7
Snellezza	$\lambda = L/\sqrt{I/A}$	11.7
Cilindro in pressione	$\sigma_c = \Delta p\, r/t,\ \sigma_l = \sigma_c/2$	11.8
Gasdinamica		
Rapporti di ristagno	$T_0/T = 1 + \frac{\gamma-1}{2} M^2$	11.9
Rapporto d'area	A/A^* (relazione d'area)	11.9
Urto normale	$p_2/p_1 = 1 + \frac{2\gamma}{\gamma+1}(M_1^2 - 1)$	11.10
Vettori (capitolo 8)		
Modulo e angolo	`np.linalg.norm`, `math.atan2`	§ 7.2
Prodotto scalare	$\mathbf{A} \cdot \mathbf{B} = AB\cos\gamma$	§ 7.6
Prodotto vettoriale	$(\mathbf{A} \times \mathbf{B})_z = A_x B_y - A_y B_x$	§ 7.7
Momento di una forza	$M_z = r_x F_y - r_y F_x$	§ 7.9
Riduzione a un polo	$\mathbf{R} = \sum \mathbf{F}_i,\ M = \sum M_i$	§ 7.12

A.15 Il prompt tecnico: modello compilabile

Un prompt efficace ha cinque componenti. Mancandone anche una sola, l'assistente riempie il vuoto con un'ipotesi propria — e sarà un'ipotesi che voi non avete scelto.

#	Componente	Domanda cui risponde
1	Modello fisico	quali equazioni, con quali ipotesi?
2	Dati con unità	quali numeri, in quali unità?
3	Dominio	entro quali limiti il modello vale?
4	Formato	come dev'essere l'output?
5	Valore di controllo	come faccio a sapere che è giusto?

Prompt pronto all'uso

*Scrivi un programma Python che … **[compito]**. Modello: … **[equazioni e ipotesi, esplicite]**. Dati: … **[valori con unità di misura]**. Dominio: … **[limiti di validità e comportamento fuori dominio]**. Formato: … **[tabella, decimali, unità di stampa]**. Controllo: … **[un valore noto, oppure una proprietà che il risultato deve possedere]**.*

La quinta componente è la più trascurata e la più importante. Non deve necessariamente essere un numero: può essere una *proprietà* verificabile — «la tensione circonferenziale deve risultare doppia della longitudinale», «la stima a peso costante deve essere un minorante», «i due prodotti scalari devono dare zero». Un prompt privo della quinta componente produce codice che nessuno può collaudare, e codice non collaudabile non è ingegneria.

A.16 Scheletro di programma

Struttura da copiare all'inizio di ogni esercizio: separa i dati dai calcoli e i calcoli dalla stampa. È la stessa organizzazione di una relazione di calcolo.

schema.py

```python
# schema.py -- [che cosa calcola, in una riga]
# Cognome Nome, classe, data
import math

# --- 1. DATI ---------------------------------------------------------
W   = 1100*9.80665      # peso [N]
S   = 16.2              # superficie alare [m^2]
rho = 1.225             # densita' ISA al mare [kg/m^3]
CLmax = 1.6             # coefficiente di portanza massimo [-]

# --- 2. CONTROLLO DEL DOMINIO ----------------------------------------
if S <= 0 or rho <= 0 or CLmax <= 0:
    raise ValueError("dati non fisici: le grandezze devono essere positive")

# --- 3. CALCOLI ------------------------------------------------------
Vs = math.sqrt(2*W/(rho*S*CLmax))

# --- 4. RISULTATI ----------------------------------------------------
print(f"Velocita' di stallo Vs = {Vs:5.2f} m/s = {Vs*3.6:5.1f} km/h")

# --- 5. VALIDAZIONE --------------------------------------------------
# controllo a mano: 2*10787/(1.225*16.2*1.6) = 679.5
```

```
#                   sqrt(679.5) = 26.07 m/s   -> coincide
```

Il blocco 5 non è decorativo: scrivere *prima* il controllo che si intende eseguire, e solo dopo guardare l'output, è ciò che distingue una verifica da una speranza.

A.17 Lista di controllo prima della consegna

	Verifica
☐	Ogni variabile ha l'unità di misura indicata in commento
☐	Gli angoli sono convertiti con `radians` prima del calcolo
☐	Si è usato `atan2` e non `atan`
☐	I confronti con zero usano una tolleranza, non `== 0`
☐	Gli estremi di `range` comprendono davvero l'ultimo valore
☐	Il dominio di validità è controllato e il caso fuori dominio gestito
☐	L'output è incolonnato, leggibile e riporta le unità
☐	Almeno un valore è validato per una via indipendente
☐	Il risultato ha l'ordine di grandezza atteso
☐	I valori impossibili ($C_L > C_{Lmax}$, tensioni oltre l'ammissibile, velocità sotto lo stallo) sono riconosciuti e segnalati
☐	La convenzione di segno di momenti e reazioni è dichiarata
☐	I grafici geometrici hanno `set_aspect("equal")`
☐	Il programma è stato rieseguito dopo l'ultima modifica

Osservazione

L'ultima riga sembra banale e non lo è: la causa più frequente di incoerenza fra il codice e la relazione è un programma modificato e non rilanciato, con l'output della versione precedente ancora sullo schermo. Rieseguite, poi copiate.

Capitolo B

Diagnostica degli errori più frequenti

B.1 Errori di sintassi (il programma non parte)

Messaggio	Diagnosi tipica
`SyntaxError: '(' was never closed`	parentesi non chiusa (contatele in coppia)
`IndentationError`	rientro mancante o disallineato dopo `if`/`for`/`def`
`SyntaxError: invalid syntax`	spesso: `=` al posto di `==` in una condizione, o `:` dimenticato

B.2 Errori a tempo di esecuzione (il programma si ferma)

Messaggio	Diagnosi tipica
`NameError: name 'Rho' is not defined`	variabile mai creata o maiuscole diverse
`TypeError: ... 'str' and 'int'`	dimenticata la conversione `float(input(...))`
`ZeroDivisionError`	denominatore nullo: quale dato lo annulla?
`ValueError: math domain error`	radice o logaritmo di numero negativo: controllare il dominio fisico
`IndexError: list index out of range`	indice oltre `len(lista)-1`: le liste partono da 0
`ModuleNotFoundError`	modulo non installato (`pip install numpy`) o nome errato

B.3 Errori silenziosi (il peggio: il programma "funziona")

Sono gli errori che non producono messaggi ma numeri sbagliati; si scovano *solo* con la validazione.

- **Unità mescolate**: piedi nei metri, nodi nei m/s, kW nei W. Sintomo: errori di fattore 0,3048, 0,5144, 1000.
- **Virgola per punto decimale**: `1,225` crea una coppia, non un numero.
- **`range` ed estremo escluso**: `range(0, 11000, 1000)` si ferma a 10 000.
- **Divisione intera indesiderata**: raro in Python 3, ma `//` al posto di `/` tronca il risultato.
- **Formula fuori dominio**: troposfera oltre 11 000 m, polare oltre lo stallo, Eulero sotto la snellezza limite. Il numero esce, la fisica no.

Attenzione

Regola finale, che riassume l'intero volume: *nessun numero è un risultato finché non ha superato un confronto indipendente* — una tabella, un calcolo manuale, un caso limite dalla soluzione nota. Vale per il codice scritto da voi, per quello scritto dall'IA e, un giorno, per quello firmato in ufficio tecnico.