Appendice

Introduzione

In questo tutorial mostriamo come realizzare una simulazione termo-fluidodinamica utilizzando strumenti open-source: `Gmsh` per la generazione della mesh, `Docker` per l'ambiente di calcolo, `FEniCSx` per la risoluzione numerica e `ParaView` per la visualizzazione dei risultati.

Prerequisiti

Per seguire correttamente il tutorial è necessario disporre dei seguenti strumenti e file:

- un **file** `.geo` di Gmsh, da cui generare la mesh del dominio;
- il **file** `.msh` esportato da Gmsh, contenente la discretizzazione e i tag di contorno;
- un **file** `.py` scritto in Python con FEniCSx, che implementa la simulazione numerica;
- un **Dockerfile**, che definisce l'ambiente software necessario per eseguire FEniCSx;
- **Docker Desktop** installato e avviato, per gestire i container ed eseguire lo script Python;
- **ParaView**, per aprire i file di output (`.xdmf`) e visualizzare i risultati.

Si raccomanda di mantenere tutti i file del progetto nella **stessa cartella** per semplificare la gestione e l'esecuzione del tutorial.

Caso di studio

Il caso di studio scelto come esempio applicativo del metodo di simulazione numerica riguarda una **cavità quadrata bidimensionale** di lato $H = 0.10\,\mathrm{m}$, completamente chiusa e riempita di acqua. Si tratta di una configurazione classica per l'analisi termo–fluidodinamica, che consente di verificare l'efficienza del modello nel risolvere un problema accoppiato di moto e trasporto termico.

La mesh del dominio, generata in `Gmsh` e salvata come `geometria.msh`, è strutturata e composta da celle quadrate di dimensioni uniformi. La simulazione è di tipo **transiente** e **incomprimibile**, condotta nel dominio del tempo a partire da condizioni iniziali di quiete e temperatura uniforme.

Condizioni al contorno

Le condizioni al contorno applicate sono le seguenti:

- **Parete inferiore:** mantenuta a temperatura costante pari a $T_{\mathrm{bottom}} = 298\,\mathrm{K}$;
- **Parete superiore:** mantenuta a temperatura costante pari a $T_{\mathrm{top}} = 293\,\mathrm{K}$;
- **Pareti laterali:** adiabatiche, ovvero con flusso termico nullo;
- **Tutte le pareti:** condizione di no–slip, cioè velocità nulla in ogni punto del contorno.

Il campo di temperatura iniziale è uniforme e pari a $T_0 = 293\,\mathrm{K}$, mentre la velocità iniziale è nulla in tutto il dominio.

Il fluido considerato è aria($T_{\mathrm{ref}} = 293\,\mathrm{K}$). I valori adottati per le proprietà fisiche sono riportati nella Tabella 4.1.

Creazione della mesh su Gmsh

La geometria del dominio è costruita mediante il software `Gmsh`. Si tratta di una cavità quadrata di lato 0.1 m, discretizzata con una mesh strutturata composta da celle quadrate regolari.

Il file di definizione della geometria, scritto in linguaggio `.geo`, è riportato di seguito:

```
// Quadrato 0.1 x 0.1 con celle QUADRATE n x n
SetFactory("OpenCASCADE");

// Parametri
L = 0.1;
H = 0.1;
n = 40; // stesse divisioni su x e y -> lato cella = 0.1/n

// Geometria
Point(1) = {0, 0, 0, 1};
Point(2) = {0, H, 0, 1};
Point(3) = {L, H, 0, 1};
Point(4) = {L, 0, 0, 1};

Line(1) = {1, 2}; // x=0
Line(2) = {2, 3}; // y=H
Line(3) = {3, 4}; // x=L
Line(4) = {4, 1}; // y=0

Curve Loop(1) = {1, 2, 3, 4};
Plane Surface(1) = {1};

// Gruppi fisici (opzionali)
Physical Curve("left", 10)  = {1};
Physical Curve("top", 11)    = {2};
Physical Curve("right", 12) = {3};
Physical Curve("bottom", 13) = {4};
Physical Surface("fluid", 14) = {1};

// Mesh strutturata quadrata
Transfinite Curve {2, 4} = n Using Progression 1; // orizzontali (lungo x)
Transfinite Curve {1, 3} = n Using Progression 1; // verticali (lungo y)
Transfinite Surface {1} = {1, 2, 3, 4};
```

Questo script definisce:

- i quattro vertici del quadrato mediante comandi `Point`;
- le quattro linee di contorno con i comandi `Line`;

- la superficie piana interna, delimitata dal `Curve Loop`;
- i `Physical Groups`, utilizzati per assegnare un nome logico alle pareti (utile per le condizioni al contorno);
- la mesh strutturata, generata tramite il comando `Transfinite`, che garantisce una suddivisione regolare in celle quadrate di dimensioni uniformi.

La mesh così definita deve essere esportata direttamente dal software `Gmsh` in formato `.msh` (versione 2) e salvata con il nome `geometria.msh`, in modo da poter essere successivamente importata dallo script `Python` per la risoluzione numerica del problema.

Creazione del DockerFile

Per garantire un ambiente riproducibile utilizziamo un `Dockerfile`, ovvero un file di testo *senza estensione* (il nome deve essere esattamente `Dockerfile`) posto nella stessa cartella del progetto

```
FROM dolfinx/dolfinx:stable
RUN python3 -m pip install --no-cache-dir meshio gmsh
WORKDIR /workspace
CMD ["bash"]
```

Spiegazione delle direttive:

- `FROM dolfinx/dolfinx:stable`: immagine base ufficiale con FEniCSx/DOLFINx preinstallato .
- `RUN python3 -m pip install ...`: installa i pacchetti Python aggiuntivi necessari al tutorial: `meshio` (lettura/scrittura mesh) e `gmsh` (API Python).
- `WORKDIR /workspace`: imposta la cartella di lavoro interna al container dove monteremo i file del progetto.
- `CMD ["bash"]`: all'avvio del container apre una shell interattiva; da qui potremo lanciare `python3`.

Docker Desktop

Per eseguire i container creati con il `Dockerfile` è necessario che **Docker Desktop** sia installato e in esecuzione.

Docker Desktop fornisce l'interfaccia grafica che gestisce i container e i volumi sul sistema operativo (Windows o macOS), permettendo di avviare, monitorare ed eliminare i container in modo semplice.

Dopo l'installazione, è sufficiente aprire Docker Desktop e verificare che l'icona nella barra delle applicazioni indichi lo stato "Running".

Se Docker Desktop non è attivo, i comandi `docker build` e `docker run` non saranno disponibili.

Nel nostro caso Docker Desktop rimane aperto in background mentre, dal terminale vengono eseguiti i comandi per costruire l'immagine e lanciare il container. Una volta avviato, Docker Desktop permette di visualizzare:

- le immagini disponibili (es. `cavita-fenicsx` create dal nostro `Dockerfile`);
- i container in esecuzione (dove lanciato lo script Python);

È importante mantenere Docker Desktop sempre attivo durante le fasi di compilazione (`docker build`) ed esecuzione (`docker run`). In questo modo l'ambiente predisposto con il `Dockerfile` sarà pienamente operativo e lo script `cavita.py` potrà essere eseguito senza problemi.

File Python

Il file `Python` costituisce la parte centrale del progetto, nella quale viene implementata la simulazione numerica.

In questo script vengono importati i pacchetti necessari, letta la mesh generata in `Gmsh`, definite le costanti del problema, costruiti gli spazi funzionali per le variabili incognite e risolto, passo dopo passo, il sistema di equazioni differenziali che governa il moto e il trasporto termico del fluido. ù

La struttura dello script segue un ordine logico suddiviso in blocchi funzionali:

1. Importazione delle librerie;
2. Importazione della mesh da `Gmsh`;
3. Definizione delle costanti e dei parametri di simulazione;
4. Costruzione degli spazi funzionali;
5. Impostazione delle condizioni iniziali e al contorno;
6. Formulazione variazionale del problema;
7. Integrazione temporale;
8. Esportazione dei risultati.

Nei paragrafi seguenti ciascun passaggio è descritto nel dettaglio.

Importazione delle librerie

```
import os
import numpy as np
from mpi4py import MPI
from petsc4py import PETSc
import ufl

from dolfinx import mesh as dmesh, fem, io
from dolfinx.io import gmshio
from dolfinx.fem.petsc import LinearProblem
from basix.ufl import element, mixed_element

comm = MPI.COMM_WORLD
rank = comm.rank
TAG = (lambda s: print(s, flush=True)) if rank == 0 else (lambda s: None)
```

Questa sezione prepara l'ambiente di lavoro importando tutte le librerie necessarie.

- **NumPy** gestisce le operazioni numeriche elementari, in particolare le manipolazioni di vettori e matrici.

- **mpi4py** e **petsc4py** forniscono il supporto alla parallelizzazione, rendendo il codice compatibile con architetture multi–processore (MPI).
- **UFL** (Unified Form Language) consente di scrivere in modo simbolico le equazioni nella forma variazionale, indipendentemente dal tipo di elemento finito utilizzato.
- **DOLFINx** costituisce il motore principale di FEniCSx: gestisce la mesh, gli spazi funzionali, la costruzione dei termini integrali e la risoluzione dei sistemi lineari.
- **Basix** definisce la natura e il grado degli elementi finiti impiegati nella discretizzazione.

Importazione della mesh da `Gmsh`

```
msh_name = "geometria.msh"
if rank == 0 and not os.path.exists(msh_name):
    raise FileNotFoundError(msh_name)
ret = gmshio.read_from_msh(msh_name, comm, gdim=2)
mesh = ret if isinstance(ret, dmesh.Mesh) else ret[0]

x = mesh.geometry.x
xmin, ymin = float(np.min(x[:,0])), float(np.min(x[:,1]))
xmax, ymax = float(np.max(x[:,0])), float(np.max(x[:,1]))
H = max(xmax-xmin, ymax-ymin)
```

In questo blocco il codice importa la mesh, precedentemente generata in `Gmsh` e salvata come `geometria.msh`.

Il file viene letto con la funzione `gmshio.read_from_msh`, che restituisce un oggetto `mesh` contenente le informazioni geometriche e topologiche del dominio.

La dimensione geometrica `gdim=2` indica che il problema è bidimensionale.

Definizione delle costanti e dei parametri numerici

```
rho0 = 1.225
mu   = 1.8e-5
nu   = mu / rho0                #      1.47e-5 m^2/s
k    = 0.025
```

```
cp    = 1005.0
alpha = k / (rho0 * cp)          #      2.03e-5 m^2/s
beta  = 3.0e-3
g     = 9.81
g_vec = ufl.as_vector((0.0, -g))
t_end = 100.0
dt    = 0.01
nsteps = int(round(t_end/dt))
save_every = 10
```

Qui vengono definiti i parametri che caratterizzano il fluido e il calcolo numerico.

Le costanti fisiche $(\rho_0, \mu, k, c_p, \beta, g)$ sono proprietà dell'acqua a temperatura ambiente. Da queste si derivano due grandezze adimensionali di rilievo numerico: ν, la viscosità cinematica, e α, la diffusività termica.

Segue la definizione dei parametri temporali della simulazione: `t_end` rappresenta la durata complessiva (100 s), `dt` è il passo temporale (0.01 s), e `nsteps` il numero totale di iterazioni temporali.

Il parametro `save_every` controlla la frequenza di salvataggio dei risultati, in modo da ridurre il numero di file scritti senza compromettere la risoluzione temporale dell'analisi.

Costruzione degli spazi funzionali

```
cell = mesh.ufl_cell().cellname()
Ve = element("Lagrange", cell, 2, shape=(2,))
Qe = element("Lagrange", cell, 1)
Te = element("Lagrange", cell, 2)
W  = fem.functionspace(mesh, mixed_element([Ve, Qe]))
V, _ = W.sub(0).collapse()
Q, _ = W.sub(1).collapse()
Th = fem.functionspace(mesh, Te)
```

Gli spazi funzionali definiscono la base numerica su cui vengono proiettate le variabili incognite. La funzione `element` stabilisce il tipo di elemento finito associato a ciascuna grandezza:

- **Velocità** (`Ve`): elementi di tipo Lagrange di grado 2 con due componenti, corrispondenti alle direzioni x e y;

- **Pressione** (`Qe`): elementi Lagrange di grado 1;
- **Temperatura** (`Te`): elementi Lagrange di grado 2.

La combinazione tra elementi quadratici per la velocità e lineari per la pressione forma la cosiddetta coppia di **Taylor–Hood**, che assicura stabilità numerica per la discretizzazione delle equazioni di Navier–Stokes incomprimibili.

La variabile `W` rappresenta lo spazio funzionale misto composto dalle due variabili $(\mathbf{u}, p)$. Le funzioni `V` e `Q` vengono poi ricavate come sottospazi di `W`, mentre `Th` è lo spazio dedicato alla temperatura.

Questa organizzazione permette di risolvere contemporaneamente il campo di velocità e pressione in un unico sistema lineare accoppiato, e di trattare la temperatura in un problema separato ma collegato.

Condizioni iniziali e al contorno

```
u_zero = fem.Function(V); u_zero.x.array[:] = 0.0
T_bottom = 298.0
T_top    = 293.0
T_init   = 293.0
T_ref    = 293.0
```

In questo blocco vengono impostate le condizioni di partenza della simulazione. Il vettore di velocità iniziale (`u_zero`) è nullo in tutto il dominio, mentre la temperatura iniziale è uniforme e pari alla temperatura di riferimento ($T_{\text{ref}} = 293\,\text{K}$).

Le condizioni al contorno sono di tipo Dirichlet per le pareti isoterme (superiore e inferiore) e di tipo Neumann omogenee (naturali) per le pareti laterali, che sono adiabatiche.

La condizione di *no-slip* per la velocità impone che ogni punto del contorno sia solidale con la parete, annullando la velocità tangenziale e normale.

Formulazione variazionale

```
a_mom = (
    (1.0/dt)*ufl.inner(U, v)*dx
    + ufl.inner(ufl.grad(U)*u_n, v)*dx
    + 2*nu*ufl.inner(ufl.sym(ufl.grad(U)), ufl.sym(ufl.grad(v)))*dx
    - ufl.inner(P, ufl.div(v))*dx
    + ufl.inner(ufl.div(U), q)*dx
)
L_mom = (1.0/dt)*ufl.inner(u_n, v)*dx - beta*ufl.inner((Tn - T_ref)*g_vec,
   v)*dx
```

Le equazioni differenziali vengono espresse nella loro forma variazionale (o debole), ossia moltiplicate per una funzione di test e integrate sul dominio.

Questo approccio è alla base del metodo degli elementi finiti e consente di gestire geometrie complesse e condizioni al contorno arbitrarie.

La prima forma (`a_mom`) rappresenta i termini del bilancio della quantità di moto, mentre la seconda (`L_mom`) costituisce il termine noto del sistema.

Il termine temporale $(1/dt)\,\mathrm{inner}(U, v)$ impone la dipendenza nel tempo, mentre i termini con il gradiente di $\mathbf{u}$ e la viscosità ν rappresentano la diffusione della quantità di moto.

Il termine $-\beta(T - T_{\mathrm{ref}})\,\mathbf{g}$ introduce la dipendenza della densità dalla temperatura.

Integrazione temporale

```
for n in range(1, nsteps+1):
    t = n*dt
    w_sol = problem_NS.solve()
    Uh, map_u = W.sub(0).collapse()
    Ph, map_p = W.sub(1).collapse()
    u_out = fem.Function(Uh); u_out.x.array[:] = w_sol.x.array[map_u]
    p_out = fem.Function(Ph); p_out.x.array[:] = w_sol.x.array[map_p]

    T_out = problem_T.solve()

    u_n.interpolate(u_out)
    p_n.interpolate(p_out)
    Tn.x.array[:] = T_out.x.array
```

Il ciclo temporale rappresenta il cuore della simulazione transiente. Ad ogni iterazione:

1. vengono risolti i campi di velocità e pressione (`problem_NS`);
2. viene risolta l'equazione della temperatura (`problem_T`);
3. i risultati vengono salvati e utilizzati come condizioni iniziali per il passo successivo.

Il tempo simulato avanza di un intervallo Δt a ogni ciclo.

L'algoritmo implementa quindi una soluzione temporale esplicita del sistema accoppiato, aggiornando le variabili primarie $(\mathbf{u}, p, T)$ fino al tempo finale.

Esportazione dei risultati

```
xdmf = io.XDMFFile(mesh.comm, "results_transient.xdmf", "w")
xdmf.write_mesh(mesh)
```

Il file di output è scritto in formato `XDMF`, compatibile con `ParaView`. Questo formato conserva sia la geometria del dominio sia i valori delle variabili ai nodi della mesh.

Al termine della simulazione il file `results_transient.xdmf` conterrà la sequenza temporale dei campi di velocità, pressione e temperatura, pronti per l'analisi e la visualizzazione.

L'intero script costituisce un esempio completo di configurazione numerica in FEniCSx, capace di risolvere un problema termofluidodinamico transiente con geometria arbitraria e condizioni al contorno personalizzabili.

Esecuzione della simulazione in ambiente `Docker`

Una volta predisposti tutti i file del progetto (`geometria.msh`, `simulazione.py` e `Dockerfile`), è possibile eseguire la simulazione all'interno di un ambiente virtualizzato tramite `Docker`. In questo modo si garantisce la completa portabilità del codice, evitando problemi di compatibilità tra versioni di librerie o sistemi operativi.

Per avviare la simulazione è necessario utilizzare il terminale **PowerShell** di Windows (o un terminale equivalente su altri sistemi operativi) e posizionarsi esattamente nella cartella che contiene tutti i file del progetto. L'intera procedura si articola in due passaggi principali:

1. **Creazione dell'immagine Docker:** questo comando costruisce un'immagine locale basata sul `Dockerfile` presente nella directory corrente.

```
docker build -t simulazione_termofluidodinamica .
```

2. **Esecuzione del container e avvio della simulazione:** una volta creata l'immagine, la simulazione può essere lanciata tramite:

```
docker run --rm -v "${PWD}:/workspace" simulazione_termofluidodinamica
    python3 /workspace/simulazione.py
```

Nel dettaglio:

- `-rm` elimina automaticamente il container al termine dell'esecuzione, evitando accumulo di risorse temporanee;
- `-v "${PWD}:/workspace"` monta la cartella locale nella directory di lavoro del container, rendendo accessibili i file del progetto all'ambiente virtuale;
- `simulazione_termofluidodinamica` è il nome dell'immagine Docker creata nel passaggio precedente;
- `python3 /workspace/simulazione.py` esegue lo script Python all'interno del container.

Al termine dell'esecuzione, nella stessa cartella di lavoro verrà generato il file di output `results_transient` contenente i campi di temperatura, pressione e velocità, successivamente visualizzabili e analizzabili in `ParaView`.

Apertura dei file. Dopo aver lanciato la simulazione con successo, nella cartella del progetto sono presenti i file:

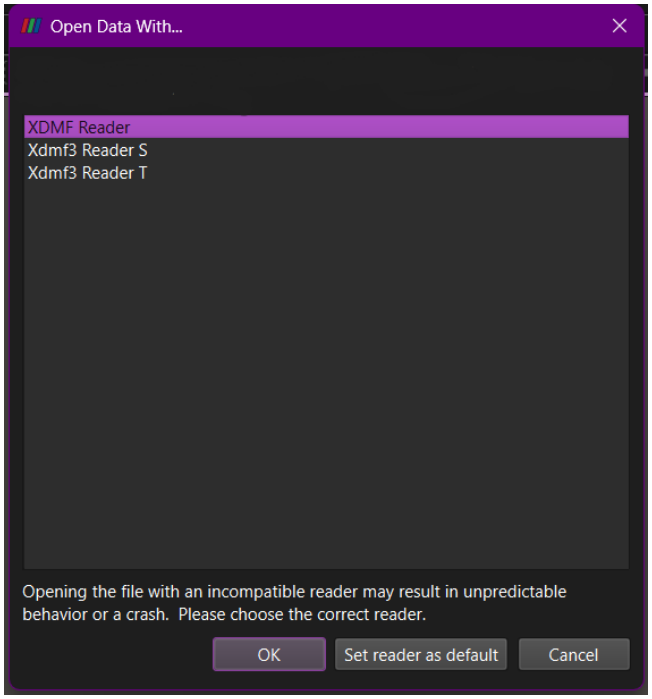
- `temperature.xdmf` — campo di temperatura,
- `velocity.xdmf` — campo di velocità,
- `pressure.xdmf` — campo di pressione.

Visualizzazione in Paraview

In ParaView è sufficiente aprire uno di questi file e, dal menu *Pipeline Browser*, selezionare il campo da visualizzare. Gli altri file possono essere caricati successivamente per effettuare analisi comparative.

ParaView riconosce automaticamente se i file contengono una serie temporale.

- Nel caso di **simulazioni stazionarie**, è sufficiente aprire il file e visualizzare il risultato al tempo unico disponibile (esecutore **S**).
- Nel caso di **simulazioni transitorie**, ParaView offre la possibilità di scorrere tra i vari time-step (esecutore **T**), creando anche animazioni dell'evoluzione temporale.



Alcuni esempi di operazioni utili:

- colorare il dominio secondo il campo scalare di interesse (es. temperatura T);
- fissare una scala di colori coerente per confronti temporali;
- esportare immagini o animazioni in formato video.

La visualizzazione con ParaView costituisce l'ultimo passo del workflow: dopo aver definito la geometria, costruito la mesh, impostato l'ambiente con Docker, e risolto numericamente il problema con FEniCSx, i risultati diventano leggibili e interpretabili in forma grafica. Questa fase consente di analizzare i campi di pressione, velocità e temperatura

Conclusione

L'intero processo dimostra come, partendo da strumenti open–source e modulari, sia possibile impostare e risolvere un problema di convezione termica in modo sistematico, riproducibile e indipendente dal sistema operativo.

La struttura del codice e dell'ambiente di calcolo presentati può essere facilmente adattata ad altri domini geometrici, fluidi o condizioni al contorno, costituendo così una base di riferimento per successive analisi e sviluppi numerici.