POLITECNICO DI MILANO

Libreria C++ per l'applicazione di metodi Immersed Boundary con particolare riferimento agli schemi ai volumi finiti centrati per il sistema delle Shallow Waters

Simone Rinco

Indice

Introduzione

I metodi di tipo *Immersed Boundary* consistono nella risoluzione numerica di un'equazione o di un sistema di equazioni a derivate parziali, tipicamente leggi di conservazione, su una griglia strutturata che é indipendente dalla forma del dominio Ω di definizione dell'equazione o del sistema di equazioni. Il principale obiettivo di questi metodi é quello di inserire il bordo del dominio, che indichiamo con Γ, sulla griglia strutturata e stabilire in che modo vengano applicate le condizioni al contorno. In questo progetto considereremo griglie cartesiane con celle quadrate e dominio rigido (indipendente dal tempo) e faremo particolare riferimento al sistema delle Shallow Waters 2D per quanto riguarda le condizioni al contorno da applicare. Nel capitolo 1 viene illustrato nel dettaglio il trattamento del dominio immerso e delle condizioni al bordo, mentre il capitolo 2 é dedicato alla descrizione del codice C++ e al suo utilizzo. La libreria é finalizzata all'impiego di metodi ai volumi finiti per i quali, nel caso più semplice, la soluzione vettoriale discreta $\mathbf{u}_{j,k}^{n+1}$ al tempo t^{n+1} che approssima le medie spaziali delle incognite $u_1(x_j, y_k, t^{n+1}), \ldots, u_N(x_j, y_k, t^{n+1})$, ossia

$$\left[\mathbf{u}_{j,k}^{n+1}\right]_i \approx u_i(x_j, y_k, t^{n+1}) = \frac{1}{\Delta x \Delta y} \int_{C_{j,k}} u_i(x, y, t^{n+1}) dx dy \qquad i = 1, \ldots, N$$

dove N é il numero delle incognite e $C_{j,k} = \left[x_{j-\frac{1}{2}}, x_{j+\frac{1}{2}}\right] \times \left[y_{k-\frac{1}{2}}, y_{k+\frac{1}{2}}\right]$, dipenda in modo esplicito dai valori della soluzione discreta al tempo t^n nelle 4 celle adicenti (Est, Ovest, Nord, Sud). In formule

$$\mathbf{u}_{j,k}^{n+1} = \mathbf{\Phi}(\mathbf{u}_{j+1,k}^{n}, \mathbf{u}_{j-1,k}^{n}, \mathbf{u}_{j,k+1}^{n}, \mathbf{u}_{j,k-1}^{n}).$$

Il sistema di equazioni modello di questi metodi consiste nella legge di conservazione

$$\mathbf{u}_t + \mathbf{f}(\mathbf{u})_x + \mathbf{g}(\mathbf{u})_y = \mathbf{0} \quad (x, y) \in \Omega,\ t > 0; \quad \mathbf{u}(x, y, t) \in \mathbb{R}^N \tag{1}$$

completato di opportune condizioni iniziali e al contorno.
Nel capitolo 2 viene descritto il possibile impiego della libreria per l'applicazione di metodi ai volumi finiti basati sulla ricostruzione polinomiale a pezzi della soluzione, per i quali $\mathbf{u}_{j,k}^{n+1}$ dipende dai valori della soluzione numerica ricostruita al tempo t^n alle 4 interfacce, ossia dai valori $\mathbf{u}^{\pm}_{j+\frac{1}{2},k}$, $\mathbf{u}^{\pm}_{j-\frac{1}{2},k}$, $\mathbf{u}^{\pm}_{j,k+\frac{1}{2}}$ e $\mathbf{u}^{\pm}_{j,k-\frac{1}{2}}$ dove per semplicità é stato omesso l'apice n relativo al tempo. Infine il capitolo 3 é dedicato alle modifiche da fare al codice sorgente per trattare diverse equazioni e/o diversi tipi di condizioni al bordo.

Listati

I listati che mostrano i codici C++ evidenziano in **grassetto** i nomi delle funzioni e dei metodi nell'intestazione della loro dichiarazione e definizione. Le keyword C++ sono evidenziate in grassetto e con colore viola (es. **typedef**), le stringhe sono evindenziate in colore blu, i commenti in arancione e i tipi non built-in in verde.

Capitolo 1

Metodi di tipo Immersed Boundary

In questo capitolo descriviamo nel dettaglio le idee chiave dei metodi *Immersed Boundary*. La prima sezione riguarda considerazioni e approssimazioni di tipo geometrico mentre la seconda si occupa del trattamento delle condizioni al bordo con particolare riferimento al caso del sistema delle Shallow Waters, nel quale le incognite sono l'altezza totale dell'acqua $Z = h + B$ dove B é l'altezza del fondale costante nel tempo, la portata $Q_x = uh$ in direzione x e la portata $Q_y = vh$ in direzione y. Le quantità u e v rappresentano le componenti della velocità dell'acqua nelle direzioni x e y.

1.1 Trattamento geometrico del bordo

Consideriamo una griglia cartesiana composta[1] da $(N_x+2)\times(N_y+2)$ celle quadrate di centro (x_j, y_k) con $j = 0, \ldots, N_x + 1$ e $k = 0, \ldots, N_y + 1$ e un dominio poligonale assegnato per punti (i vertici). La prima fase per applicare un qualsiasi metodo di tipo *Immersed Boundary* consiste nell'adeguamento del bordo alla griglia. Questa fase può causare una leggera modifica del dominio originale che diventa tanto più trascurabile quanto fine diventa il passo di griglia $\Delta x = \Delta y$. Chiamiamo *dominio computazionale*, per distinguerlo dal dominio reale, il dominio costruito in questa fase e definiamo *cella tagliata* una cella che risulti attraversata dal bordo del dominio computazionale.

Il dominio computazionale deve rispettare tre regole fondamentali:

1. ogni cella tagliata é attraversata da un unico segmento rettilineo;
2. la linea che unisce i centri di due celle adiacenti[2] attraversa al più una volta il dominio computazionale;
3. se due celle adiacenti sono tagliate i segmenti che tagliano le due celle si devono incontrare in un punto.

Queste regole permettono di manipolare efficacemente domini complessi limitando il numero di configurazioni possibili per l'assegnamento delle condizioni al bordo, discusse nella sezione successiva. Con riferimento alla fluidodinamica[3] consideriamo *celle bagnate* quelle celle il cui centro sia all'interno del dominio computazionale, mentre *celle asciutte* quelle con centro all'esterno del dominio computazionale.

[1] Utilizziamo qui la stessa notazione impiegata dal codice C++.

[2] Data una certa cella, le celle ad essa adiacenti sono le 4 celle a Est, a Ovest, a Nord e a Sud. Due celle sono quindi adiacenti se condividono uno dei 4 lati di bordo cella.

[3] Questa sezione è in realtà di carattere generale. Le celle bagnate sono le celle nelle quali si vuole calcolare la soluzione numerica.

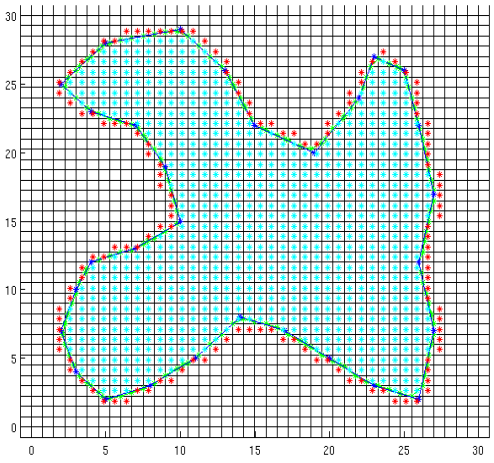

Figura 1.1: Esempio di adattamento del dominio alla griglia. In azzurro: wet cells; in rosso: ghost cells; in blu: dominio reale; in verde: dominio computazionale.

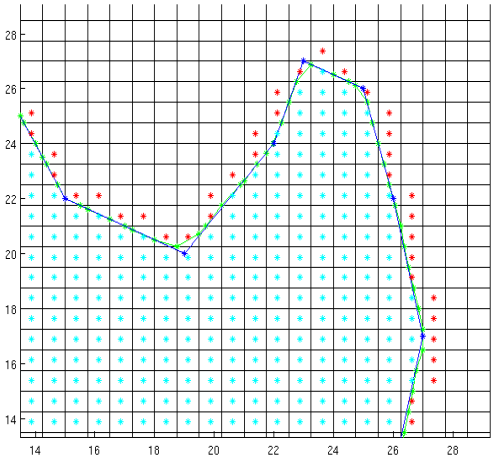

Figura 1.2: Zoom della figura precedente sull'angolo in alto a destra: si nota che il dominio computazionale é leggermente diverso dal dominio reale.

Nei metodi *Immersed Boundary* risulta fondamentale il concetto di ghost cell.

Definizione 1 (Ghost cell) *Una ghost cell é una cella asciutta che ha almeno una cella adiacente bagnata.*

Il dominio reale può essere definito, anzichè per punti, attraverso una curva di livello. Sia $f(x, y)$ una funzione differenziabile con gradiente non nullo negli zeri di f, ossia nei punti (x^*, y^*) tali che $f(x^*, y^*) = 0$. Allora il dominio reale è definito dalla relazione

$$\Omega = \{(x, y) \in \mathbb{R}^2 : f(x, y) < 0\} \tag{1.1}$$

e si ha

$$\Gamma = \partial\Omega = \{(x, y) \in \mathbb{R}^2 : f(x, y) = 0\}.$$

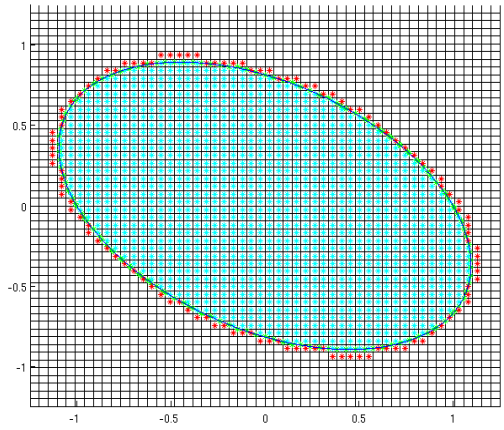

Figura 1.3: Esempio di adattamento del dominio alla griglia nel caso *level set*.

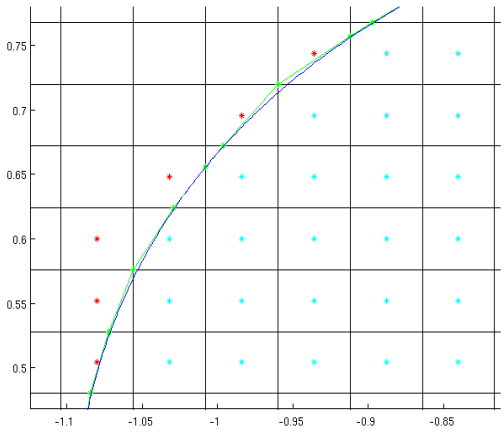

Figura 1.4: Particolare della figura precedente: si nota anche in questo caso che il dominio computazionale é leggermente diverso dal dominio reale.

Nella figura 1.3 il dominio è definito come in 1.1 con

$$f(x, y) = 2x^2 + 3y^2 + 2xy - 2.$$

1.2 Trattamento delle condizioni al bordo

Un classico metodo ai volumi finiti richiede la conoscenza delle medie di cella nelle 4 celle adiacenti a quella corrente, come descritto nell'introduzione. Di fatto si deve iterare sulle celle bagnate, che sono le uniche nelle quali interessa calcolare la soluzione, e applicare lo schema desiderato. Nel caso in cui una o più tra celle adiacenti a quella considerata sia asciutta (e quindi è una ghost cell) si devono assegnare a queste dei valori fittizi in modo da rispettare in qualche senso le condizioni al bordo. Si procede poi con l'applicazione dello schema numerico come se tutte le celle adiacenti fossero wet cells. In questa sezione descriviamo nel dettaglio come imporre i valori alle ghost cells con riferimento al caso delle Shallow Waters.
Le condizioni al bordo su Γ sono le seguenti[4]:

$$Q_n = 0, \tag{1.2}$$

$$\frac{dQ_t}{dn} = 0, \tag{1.3}$$

$$\frac{dZ}{dn} = 0. \tag{1.4}$$

Le quantità Q_n e Q_t sono rispettivamente le componenti normale e tangente al bordo della portata. Se $\mathbf{n} = (n_x, n_y)$ é il versore normale uscente al bordo e $\mathbf{t} = (t_x, t_y) = (-n_y, n_x)$ il versore tangente si ha

$$Q_n = Q_x n_x + Q_y n_y \, , \, Q_t = Q_x t_x + Q_y t_y.$$

Per assegnare un valore fittizio al centro delle ghost cells si proietta tale punto, che chiamiamo *ghost point* (GP), all'interno del dominio in direzione normale al bordo individuando il *boundary point* (BP) sul bordo e il *reflected point* (RP) all'interno del dominio, in modo che la distanza tra GP e BP sia la stessa tra BP e RP. Successivamente si calcola il valore dell'incognita voluta nel reflected point tramite un particolare interpolante e se ne estrapola linearmente il valore nel ghost point sulla base delle condizioni al bordo desiderate. Per ricostruire il valore in RP si utilizzano, laddove possibile, i valori delle incognite nel quadrato di interpolazione così definito:

Definizione 2 *Dato un RP si definisce quadrato di interpolazione*[5] *del RP l'insieme dei quattro centri cella che sono i vertici del più piccolo quadrato che contiene RP avente per vertici dei centri cella.*

Con le regole elencate precedentemente si possono presentare solo 3 situazioni possibili che danno luogo a diverse procedure per l'imposizione delle condizioni al bordo:

1. tutti i vertici di IS sono centri cella bagnati (caso *all_wet*);
2. IS contiene uno o più ghost points mentre gli altri vertici sono bagnati (caso *GPs*);
3. IS contiene uno e un solo centro cella asciutto che non è un ghost point (caso *dry_no_GP*).

1.2.1 Caso 1: *all_wet*

Il caso *all_wet* in cui tutti i vertici del quadrato di interpolazione sono bagnati é il più semplice per due motivi: la procedura di calcolo del valore in RP é la stessa per ogni variabile e non dipende dalle condizioni al bordo implementate. Inoltre é l'unico caso che prescinde dal particolare sistema di equazioni considerato. Indichiamo con ϕ una generica incognita della

[4] Condizioni di bordo chiuso.
[5] Di seguito indicato con IS.

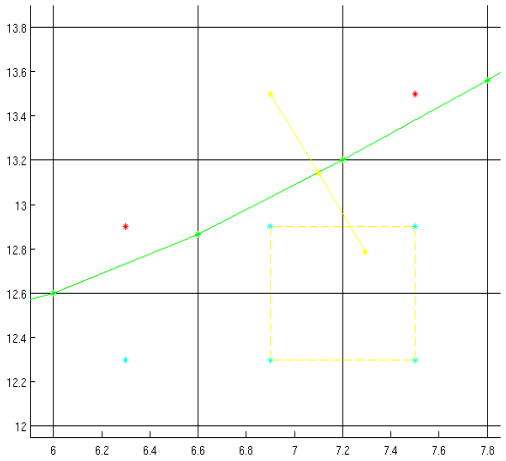

Figura 1.5: Situazione *all wet* nella quale ogni vertice del quadrato di interpolazione é bagnato. In giallo sono rappresentati il GP (centro della ghost cell), il BP e il RP.

quale sono disponibili i valori in tutti i vertici di IS e numeriamo i vertici di IS da 1 a 4 partendo dal vertice di sud-ovest e procedendo in senso antiorario. Con ovvio significato della notazione usata si deve calcolare ϕ_{RP} noti i valori ϕ_i per $i = 1, 2, 3, 4$. Supponendo ϕ della forma

$$\phi(x, y) = c_1 xy + c_2 x + c_3 y + c_4$$

si trovano i coefficienti di interpolazione c_i risolvendo il sistema $M\mathbf{c} = \mathbf{\Phi}$ con

$$M = \begin{bmatrix} x_1y_1 & x_1 & y_1 & 1 \\ x_2y_2 & x_2 & y_2 & 1 \\ x_3y_3 & x_3 & y_3 & 1 \\ x_4y_4 & x_4 & y_4 & 1 \end{bmatrix}$$

e $[\mathbf{c}]_i = c_i$, $[\mathbf{\Phi}]_i = \phi_i$ mentre x_i e y_i sono le coordinate dell'i-esimo vertice (centro cella) di interpolazione. Osserviamo che la matrice M é la stessa per ognuna delle incognite. Una volta noti i coefficienti c_i si calcola

$$\phi_{RP} = c_1 x_{RP} y_{RP} + c_2 x_{RP} + c_3 y_{RP} + c_4.$$

Infine il valore ϕ_{GP} si calcolerà in funzione delle condizioni al bordo che si vogliono implementare in quella zona del dominio (vedere più avanti).

1.2.2 Caso 2: *GPs*

Con la sigla *GPs* indichiamo la presenza in IS di almeno un GP e nessun vertice di IS che sia un centro cella asciutto ma non un ghost point. Stavolta il calcolo delle incognite in RP dipende non solo dall'incognita considerata, ma anche dalle condizioni al bordo. Faremo quindi riferimento al caso del sistema delle Shallow Waters. Nella situazione di figura 1.6 i vertici di interpolazione 2 e 3 sono bagnati e quindi sono disponibili i valori di tutte le incognite, mentre i vertici 1 e 4 sono inutilizzabili. L'idea é quella di sostituire ogni vertice inutilizzabile con un boundary point nel quale si impongono le condizioni al bordo volute. Indichiamo con BP_1 il boundary point che sostituisce il vertice 1 e con BP_4 il boundary point che sostituisce il vertice 4.

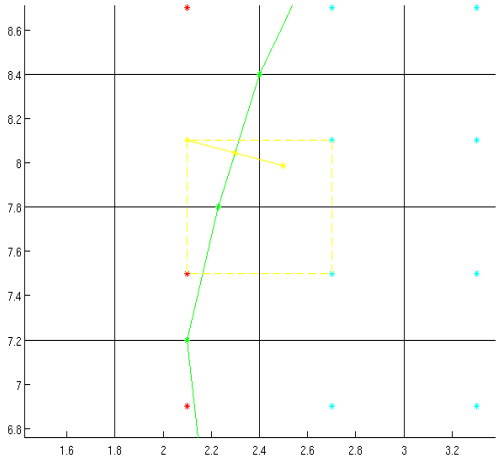

Figura 1.6: Situazione *GPs* nella quale IS contiene due vertici che sono ghost points (il ghost point stesso in giallo e il vertice di sud-ovest) e due vertici bagnati.

Caso *GPs*: altezza totale

Supponendo, come prima, che Z sia della forma

$$Z(x,y) = c_1xy + c_2x + c_3y + c_4.$$

i vertici bagnati forniscono ancora le due equazioni

$$c_1x_iy_i + c_2x_i + c_3y_i + c_4 = Z_i \quad i = 2, 3.$$

In un generico boundary point applichiamo la condizione al contorno 1.4 per l'altezza totale che porta all'equazione

$$\frac{dZ}{dn}\Big|_{BP} = \nabla Z \cdot \mathbf{n} = \nabla(c_1xy + c_2x + c_3y + c_4)|_{BP} \cdot \mathbf{n} = (c_1y_{BP} + c_2)n_x + (c_1x_{BP} + c_3)n_y = 0.$$

Si é quindi giunti ad un sistema lineare $M\mathbf{c} = \mathbf{b}$ con

$$M = \begin{bmatrix} y_{BP_1}n_{x_1} + x_{BP_1}n_{y_1} & n_{x_1} & n_{y_1} & 0 \\ x_2y_2 & x_2 & y_2 & 1 \\ x_3y_3 & x_3 & y_3 & 1 \\ y_{BP_4}n_{x_4} + x_{BP_4}n_{y_4} & n_{x_4} & n_{y_4} & 0 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \text{ e } \quad \mathbf{b} = \begin{bmatrix} 0 \\ Z_2 \\ Z_3 \\ 0 \end{bmatrix}.$$

Caso *GPs*: portata

Sempre in riferimento al caso illustrato in figura 1.7 utilizziamo i punti riflessi 1 e 4 per trovare le equazioni mancanti, sfruttando le condizioni al bordo 1.2 e 1.3 che permettono di calcolare i coefficienti di interpolazione. Supponiamo Q_x e Q_y della forma

$$Q_x(x,y) = c_{1_x}xy + c_{2_x}x + c_{3_x}y + c_{4_x}\,,\; Q_y(x,y) = c_{1_y}xy + c_{2_y}x + c_{3_y}y + c_{4_y}$$

e scriviamo il vettore dei coefficienti di interpolazione $\mathbf{c}$ nella forma

$$\mathbf{c} = \begin{bmatrix} c_{1_x} & c_{1_y} & c_{2_x} & c_{2_y} & c_{3_x} & c_{3_y} & c_{4_x} & c_{4_y} \end{bmatrix}^T.$$

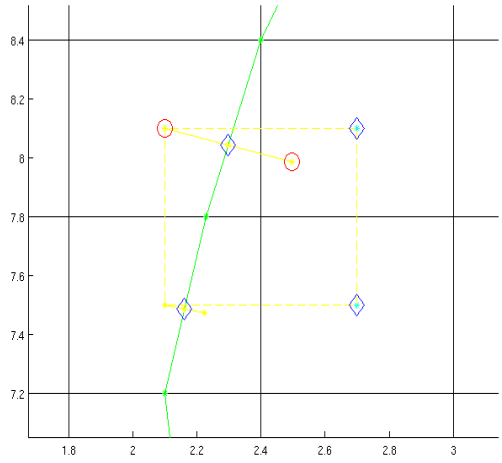

Figura 1.7: I due vertici di interpolazione 1 e 4 sono ghost points e vengono quindi sostituiti da due BP. I vertici del nuovo quadrilatero di interpolazione sono evidenziati in blu e permetteranno di calcolare il valore dell'incognita nel RP in rosso col quale si calcolerà il valore nel GP selezionato (anch'esso in rosso).

I centri cella bagnati 2 e 3 forniscono le quattro equazioni

$$c_{1_\theta} x_i y_i + c_{2_\theta} x_i + c_{3_\theta} y_i + c_{4_\theta} = (Q_\theta)_i \qquad i = 2, 3 \quad \theta = x, y.$$

Le restanti quattro equazioni si determinano imponendo le condizioni 1.2 e 1.3 nei punti riflessi 1 e 4. Dato un generico punto riflesso la condizione 1.2 porta all'equazione

$$\begin{aligned} Q_{n_{BP}} &= Q_{x_{BP}} n_{x_{BP}} + Q_{y_{BP}} n_{y_{BP}} = \\ &= (c_{1_x} xy + c_{2_x} x + c_{3_x} y + c_{4_x}) n_x + (c_{1_y} xy + c_{2_y} x + c_{3_y} y + c_{4_y}) n_y = 0 \end{aligned}$$

dove nell'ultima formula sono stati omessi i pedici BP alle coordinate x e y. La condizione 1.3 porta all'equazione, omettendo ancora i pedici BP ed indicando con $\mathbf{Q} = (Q_x, Q_y)$ il vettore portata,

$$\begin{aligned} \frac{dQ_t}{dn} &= \nabla(\mathbf{Q} \cdot \mathbf{t}) \cdot \mathbf{n} = \nabla(-Q_x n_y + Q_y n_x) \cdot \mathbf{n} = \\ &= \nabla \left(- \left(c_{1_x} xy + c_{2_x} x + c_{3_x} y + c_{4_x} \right) n_y + \left(c_{1_y} xy + c_{2_y} x + c_{3_y} y + c_{4_y} \right) n_x \right) \cdot \mathbf{n} = \\ &= \left(- \left(c_{1_x} y + c_{2_x} \right) n_y + \left(c_{1_y} y + c_{2_y} \right) n_x \right) n_x + \left(- \left(c_{1_x} x + c_{3_x} \right) n_y + \left(c_{1_y} x + c_{3_y} \right) n_x \right) n_y = \\ &= 0. \end{aligned}$$

La matrice M di interpolazione diventa quindi

$$M = \begin{bmatrix} (xyn_x)_1 & (xyn_y)_1 & (xn_x)_1 & (xn_y)_1 & (yn_x)_1 & (yn_y)_1 & (n_x)_1 & (n_y)_1 \\ (-yn_x n_y - xn_y^2)_1 & (yn_x^2 + xn_x n_y)_1 & (-n_x n_y)_1 & (n_x^2)_1 & (-n_y^2)_1 & (n_x n_y)_1 & 0 & 0 \\ x_2 y_2 & 0 & x_2 & 0 & y_2 & 0 & 1 & 0 \\ 0 & x_2 y_2 & 0 & x_2 & 0 & y_2 & 0 & 1 \\ x_3 y_3 & 0 & x_3 & 0 & y_3 & 0 & 1 & 0 \\ 0 & x_3 y_3 & 0 & x_3 & 0 & y_3 & 0 & 1 \\ (xyn_x)_4 & (xyn_y)_4 & (xn_x)_4 & (xn_y)_4 & (yn_x)_4 & (yn_y)_4 & (n_x)_4 & (n_y)_4 \\ (-yn_x n_y - xn_y^2)_4 & (yn_x^2 + xn_x n_y)_4 & (-n_x n_y)_4 & (n_x^2)_4 & (-n_y^2)_4 & (n_x n_y)_4 & 0 & 0 \end{bmatrix}$$

dove il pedice 1 nella prima e seconda riga si riferisce al BP_1 e il pedice 4 nella settima e ottava riga si riferisce al BP_4. Il termine noto assume la forma

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ (Q_x)_2 \\ (Q_y)_2 \\ (Q_x)_3 \\ (Q_y)_3 \\ 0 \\ 0 \end{bmatrix}.$$

Trovato il vettore dei coefficienti $\mathbf{c} = M^{-1}\mathbf{b}$ si calcolano finalmente

$$(Q_x)_{RP} = c_{1_x} x_{RP} y_{RP} + c_{2_x} x_{RP} + c_{3_x} y_{RP} + c_{4_x}$$

e

$$(Q_y)_{RP} = c_{1_y} x_{RP} y_{RP} + c_{2_y} x_{RP} + c_{3_y} y_{RP} + c_{4_y}.$$

1.2.3 Caso 3: *dry_no_GP*

Con la sigla *dry_no_GP* intendiamo la situazione nella quale esiste esattamente un vertice asciutto in IS che non sia il centro di una ghost cell. In questo caso non é possibile applicare le condizioni al bordo al punto riflesso del centro cella dry_no_GP che di fatto non viene nemmeno calcolato poichè la cella non é una ghost cell. Si hanno dunque solo tre vertici di interpolazione disponibili che sono centri cella bagnati o ghost points. Come nel caso precedente il calcolo delle incognite nel punto riflesso dipende dall'incognita considerata e dal tipo di condizioni al bordo. Riferendoci alle condizioni 1.2, 1.3, 1.4 per il sistema delle Shallow Waters esaminiamo separatamente il caso dell'altezza totale e della portata.

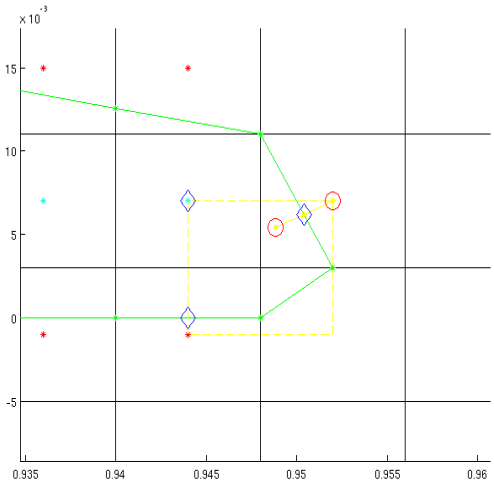

Figura 1.8: L'IS del RP relativo al GP esaminato (entrambi in rosso) ha il secondo vertice *dry_no_GP*. I tre vertici di interpolazione (in blu) sono dunque il BP relativo al primo vertice di IS, il BP relativo al terzo vertice di IS e il quarto vertice di IS.

Caso *dry_no_GP*: altezza totale

Avendo a disposizione solo tre punti di interpolazione consideriamo l'interpolante di Z della forma

$$Z(x,y) = c_1 x + c_2 y + c_3.$$

Riferendoci al caso particolare di figura 1.8 il quarto vertice é bagnato e fornisce quindi l'equazione

$$c_1 x_4 + c_2 y_4 + c_3 = Z_4.$$

Se un vertice é un GP consideriamo[6] il suo BP e applichiamo la condizione al bordo 1.4 per ottenere l'equazione

$$\left.\frac{dZ}{dn}\right|_{BP} = \nabla Z \cdot \mathbf{n} = \nabla \left(c_1 x + c_2 y + c_3\right)|_{BP} \cdot \mathbf{n} = c_1 n_x + c_2 n_y = 0.$$

La matrice M di interpolazione, il vettore $\mathbf{c}$ dei coefficienti e il termine noto $\mathbf{b}$ diventano

$$M = \begin{bmatrix} n_{x_1} & n_{y_1} & 0 \\ n_{x_3} & n_{y_3} & 0 \\ x_4 & y_4 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \text{ e } \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ Z_4 \end{bmatrix}$$

Caso *dry_no_GP*: portata

Consideriamo gli interpolanti di Q_x e Q_y della forma

$$Q_x(x,y) = c_{1_x} x + c_{2_x} y + c_{3_x} \ , \ Q_y(x,y) = c_{1_y} x + c_{2_y} y + c_{3_y}.$$

Sempre in riferimento alla situazione di figura 1.8 il quarto vertice fornisce le equazioni

$$c_{1_x} x_4 + c_{2_x} y_4 + c_{3_x} = (Q_x)_4 \ , \ c_{1_y} x_4 + c_{2_y} y_4 + c_{3_y} = (Q_y)_4.$$

Preso invece un BP e applicando la condizione al bordo 1.2 si ottiene l'equazione

$$Q_n = Q_x n_x + Q_y n_y = (c_{1_x} x + c_{2_x} y + c_{3_x}) n_x + (c_{1_y} x + c_{2_y} y + c_{3_y}) n_y = 0$$

dove per semplicità sono stati omessi i pedici ad x, y, n_x e n_y per indicare il riferimento al BP. La condizione 1.3 porta invece all'equazione

$$\begin{aligned} \frac{dQ_t}{dn} &= \nabla(\mathbf{Q} \cdot \mathbf{t}) \cdot \mathbf{n} = \nabla(-Q_x n_y + Q_y n_x) \cdot \mathbf{n} = \\ &= \left(-c_{1_x} n_y + c_{1_y} n_x\right) n_x + \left(-c_{2_x} n_y + c_{2_y} n_x\right) n_y = 0. \end{aligned}$$

nel boundary point. In definitiva la matrice M, il vettore $\mathbf{c}$ e il termine noto $\mathbf{b}$ diventano

$$M = \begin{bmatrix} (xn_x)_1 & (xn_y)_1 & (yn_x)_1 & (yn_y)_1 & (n_x)_1 & (n_y)_1 \\ -(n_x n_y)_1 & (n_x^2)_1 & -(n_y^2)_1 & (n_x n_y)_1 & 0 & 0 \\ (xn_x)_3 & (xn_y)_3 & (yn_x)_3 & (yn_y)_3 & (n_x)_3 & (n_y)_3 \\ -(n_x n_y)_3 & (n_x^2)_3 & -(n_y^2)_3 & (n_x n_y)_3 & 0 & 0 \\ x_4 & 0 & y_4 & 0 & 1 & 0 \\ 0 & x_4 & 0 & y_4 & 0 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_{1_x} \\ c_{1_y} \\ c_{2_x} \\ c_{2_y} \\ c_{3_x} \\ c_{3_y} \end{bmatrix} \text{ e } \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ (Q_x)_4 \\ (Q_y)_4 \end{bmatrix}$$

dove i pedici 1 e 3 in M si riferiscono ai BP e il pedice 4 si riferisce al centro cella bagnato. Trovati i coefficienti di interpolazione si ricavano $(Q_x)_{RP}$ e $(Q_y)_{RP}$.

[6] In realtà un GP può avere due BP (e quindi due RP) distinti. In tal caso considereremo il punto riflesso più vicino a IS.

1.2.4 Determinazione delle incognite nei ghost points

Una volta noto il valore delle incognite nel punto riflesso si esegue un'estrapolazione lineare sfruttando le condizioni al bordo. Per l'altezza totale si ha

$$\left.\frac{dZ}{dn}\right|_{BP} \approx \frac{Z_{GP} - Z_{RP}}{2d}$$

dove d é la distanza tra GP e BP e tra BP e RP. La condizione 1.4 porta dunque ad imporre

$$Z_{GP} = Z_{RP}.$$

Per la portata in direzione normale al bordo si ha, estrapolando linearmente,

$$(Q_n)_{BP} \approx \frac{(Q_n)_{GP} + (Q_n)_{RP}}{2}$$

e quindi per la 1.2 si ha

$$(Q_n)_{GP} = -(Q_n)_{RP}.$$

Infine

$$\left.\frac{dQ_t}{dn}\right|_{BP} \approx \frac{(Q_t)_{GP} - (Q_t)_{RP}}{2d}$$

e la 1.3 porta a scrivere

$$(Q_t)_{GP} = (Q_t)_{RP}.$$

Noti i valori $(Q_n)_{GP}$ e $(Q_t)_{GP}$ si calcolano i valori $(Q_x)_{GP}$ e $(Q_y)_{GP}$ risolvendo il sistema

$$\begin{bmatrix} n_x & n_y \\ -n_y & n_x \end{bmatrix} \begin{bmatrix} (Q_x)_{GP} \\ (Q_y)_{GP} \end{bmatrix} = \begin{bmatrix} (Q_n)_{GP} \\ (Q_t)_{GP} \end{bmatrix}.$$

Capitolo 2

Descrizione ed utilizzo del codice

In questo capitolo descriviamo l'installazione e l'utilizzo della libreria `libgrid.a` per l'applicazione del metodo ai volumi finiti desiderato con dominio immerso. La libreria contiene l'implementazione di varie classi tra le quali la classe `grid` che si occupa della trattazione completa della griglia e del bordo e fornisce tutti i metodi necessari ad una facile e veloce implementazione del proprio metodo. Come più volte detto la libreria é stata costruita in riferimento al sistema delle Shallow Waters, in particolare riguardo alla costruzione delle matrici di interpolazione necessarie al calcolo dei valori delle incognite nei ghost points. Il capitolo successivo illustra le modifiche da fare per trattare diversi sistemi di equazioni con diverse condizioni al bordo.

2.1 Installazione

La cartella `Immersed_boundary` contiene 7 sottocartelle e un `Makefile`. Per creare il proprio progetto e installare la libreria aprire il `Makefile` ed assegnare il nome del progetto alla variabile `PROJECT_DIR` il cui valore di default é *./my_project*. Aprire quindi un terminale e lanciare il comando `make` che provvede alla compilazione dei sorgenti della libreria e alla creazione della cartella col nome del proprio progetto e tutte le sottocartelle necessarie. Vengono copiati i file `input_parameters.pot` e `vertices.pot` dalla cartella `input_files`, viene copiata la cartella `m_files` nella cartella del proprio progetto e viene costruita la cartella `data` dove verranno salvati gli output generati dal proprio eseguibile che dovrà essere situato nella cartella `Immersed_boundary/my_project`. La libreria `libgrid.a` si trova nella cartella `Immersed_boundary/lib` mentre tutti gli include files sono nella cartella `Immersed_boundary/include`.

2.2 La classe `grid`

2.2.1 Costruttori

Il nucleo fondamentale della libreria `libgrid.a` é costituito dalla classe `grid`. Come visto nel capitolo precedente il dominio può essere assegnato tramite i vertici di un poligono o tramite level set. Vengono forniti due costruttori a seconda del modo nel quale si intende definire il dominio. Il listato 2.2 illustra il funzionamento dei due costruttori. Vengono definite le funzioni `quadratic` e `grad_quadratic` che implementano rispettivamente la funzione $f(x, y) = x^2 + y^2 - 1$ e il suo gradiente $\nabla f(x, y) = [2x\ 2y]^T$. La classe `Point2d<T>` é una semplice ed intuitiva classe per la gestione dei punti in due dimensioni. Il namespace `IMMERSED_BOUNDARY` é provvisto alcuni utili typedef: il primo é utile per la gestione delle celle della griglia, che vengono individuate mediante gli indici di riga e colonna mentre gli altri tre

```
typedef Point2d<unsigned int> label;

typedef bool (*p_comp)(const label&, const label&);
typedef double (*p_fun)(const Point2d<double>&);
typedef Point2d<double> (*p_grad_fun)(const Point2d<double>&);
```

Listing 2.1: Typedef nel namespace `IMMERSED_BOUNDARY`

sono typedef di puntatori a funzione. Il primo di questi serve per specificare[1] l'ordinamento tra labels, gli altri due per specificare rispettivamente la funzione che identifica il level set e il suo gradiente. Il namespace `IMMERSED_BOUNDARY` fornisce anche la struct `stencil_values` utile per il trattamento dei valori assunti da una certa quantità al centro e nei quattro punti cardinali rispetto ad una cella di riferimento

```
struct stencil_values // stencil_values u
{
   // fissata una cella (j,k):
   double& central; // u.central --> u(j,k)
   double& E; // u.E --> u(j+1,k)
   double& W;
   double& N;
   double& S; // u.S --> u(j,k-1)
};
```

con il relativo costruttore

```
stencil_values(double& c, double& e, double& w, double& n, double& s);
```

La funzione `main` del listato 2.2 definisce il puntatore `p_c` e lo inizializza con la funzione `IMMERSED_BOUNDARY::compare` che ordina le celle in senso lessicografico[2]. Le signature dei due costruttori e del metodo pubblico `writeout_domain` sono mostrate nel listato 2.3. Il costruttore che assembla il dominio dato per vertici riceve 3 parametri ognuno dei quali ha un valore di default. L'oggetto G2 viene quindi costruito in maniera identica all'oggetto G1. Per il parsing dei files di input viene usato `GetPot`[3]. Tramite il file `input_parameters` vengono impostati i valori `Nx` e `Ny` pari al numero di celle del dominio rettangolare che conterrà il dominio computazionale e gli estremi `x_min`, `x_max` e `y_min` di tale dominio. L'estremo `y_max` é automaticamente calcolato poichè le celle devono essere quadrate. Le celle effettivamente create sono N_x+2 e N_y+2 aggiungendo uno strato di celle all'esterno del dominio rettangolare. Per evitare di creare celle tagliate con un'area bagnata (od asciutta) troppo piccola si impedisce ai vertici dei lati del dominio computazionale di cadere in una zona troppo vicina agli spigoli della cella tagliata. Come visto ogni vertice del lato appartenente ad una cella tagliata deve cadere sul bordo della cella stessa. Il parametro `k` é un numero compreso tra 0 e 1 che esprime, in percentuale rispetto alla lunghezza dei lati, la distanza minima verticale e orizzontale dei vertici dei lati dai bordi cella. Ad esempio se $k = \frac{\sqrt{2}}{10}$ l'area più piccola delle due parti che dividono la cella tagliata é maggiore o uguale all'1% dell'area dell'intera cella quadrata.
Il dominio reale (poligonale) si ottiene congiungendo con segmenti i vertici, le cui coordinate sono contenute nell'apposito file, dal primo all'ultimo e chiudendo il dominio con il segmento avente per vertici l'ultimo e il primo.
Se si intende costruire il dominio tramite un level set si devono fornire i puntatori a funzione della funzione che definisce il level set e del suo gradiente. Anche in questo caso il file dei parametri e il puntatore della funzione che determina l'ordinamento delle celle hanno un valore

[1] Consultare, ad esempio, la guida a `map` all'indirizzo *http://www.cplusplus.com/reference/stl/map/map/*
[2] $(i,j) \preceq (k,l)$ se $i \leq k$ oppure se $i = k$ e $j \leq l$.
[3] *http://getpot.sourceforge.net/*

```cpp
#include "grid.hpp"

double quadratic(const Point2d<double>& P)
{
    double x = P(0);
    double y = P(1);

    return x*x + y*y - 1;
}

Point2d<double> grad_quadratic(const Point2d<double>& P)
{
    double x = P(0);
    double y = P(1);

    double grad_x = 2*x;
    double grad_y = 2*y;

    Point2d<double> grad(grad_x, grad_y);

    return grad;
}

int main(int argc, char **argv)
{

    IMMERSED_BOUNDARY::p_comp p_c = IMMERSED_BOUNDARY::compare;
    IMMERSED_BOUNDARY::p_fun p_f = quadratic;
    IMMERSED_BOUNDARY::p_grad_fun p_gf = grad_quadratic;

    std::string input_file("./input_parameters.pot");
    std::string vertices_file("vertices.pot");
    bool save_interp_matrices = true;

    grid G1; // costruzione dominio "per vertici"
    G1.writeout_domain(); // => G1.writeout_domain(true);

    grid G2(input_file, vertices_file, p_c); // come G1
    G2.writeout_domain(save_interp_matrices);

    grid G3(p_f, p_gf); // costruzione dominio per "level set"

    grid G4(p_f, p_gf, input_file, p_c); // come G3

    return 0;
}
```

Listing 2.2: Costruzione di oggetti di classe `grid` e metodo `writeout_domain`.

```cpp
grid(const std::string& input_f = "./input_parameters.pot", \
    const std::string& vertices_f = "./vertices.pot", \
    const p_comp& comp_function = IMMERSED_BOUNDARY::compare);
grid(const IMMERSED_BOUNDARY::p_fun& p_f, \
    const IMMERSED_BOUNDARY::p_grad_fun& p_grad_f, \
    const std::string& input_f = "./input_parameters.pot", \
    const p_comp& comp_function = IMMERSED_BOUNDARY::compare);

void writeout_domain(const bool& save_interp_matrices = true);
```

Listing 2.3: Signature dei costruttori e del metodo `writeout_domain`.

di default. I domini reali degli oggetti G3 e G4 nel listato 2.2 sono quindi pari al cerchio unitario e costruiti nello stesso modo.

2.2.2 Il metodo `writeout_domain`

Questo metodo salva le caratteristiche del dominio e le matrici di interpolazione (se viene passato il valore `true`) rispettivamente nella cartella `./data/domain` e nella cartella `./data/domain/interp_matrices`. Le matrici di interpolazione vengono salvate a seconda dei casi visti nel precedente capitolo. Poichè ogni matrice é associata ad un quadrato di interpolazione, il file contenente la matrice ha lo stesso nome del file contente l'etichetta della cella il cui centro é il vertice di sud-ovest del quadrato di interpolazione. I nomi dei file sono progressivi partendo dal numero 0. I file contenenti le caratteristiche del dominio sono:

- `cutted_cells`: contiene le labels delle celle tagliate, siano esse asciutte o bagnate. La cella alla riga i-esima é tagliata dal lato i-esimo;
- `edges`: contiene le coordinate dei vertici dei lati che formano il dominio computazionale. La riga $i-$esima del file contiene nell'ordine le coordinate x e y del vertice sinistro e le coordinate x e y del vertice destro del lato i-esimo del dominio computazionale. Per vertice destro e sinistro di un lato si intende affermare che il vertice destro del lato i coincide col vertice sinistro del lato $i+1$ per $i = 1, \ldots, N_e - 1$ dove N_e é il numero dei lati e il vertice destro del lato N_e coincide col vertice sinistro del lato 1;
- `real_vertices` (solo nel caso *vertices*): contiene i vertici del dominio reale. L'ultimo vertice di questo file coincide con il primo;
- `initial_edges` (solo nel caso *level set*): ogni riga contiene la label della cella tagliata e le coordinate dei vertici del lato nella prima fase di costruzione del dominio computazionale. Il dominio finale verrà costruito assemblando questi lati;
- `wet_cells`: contiene le labels delle celle bagnate;
- `x`: contiene le coordinate x dei bordi cella: la cella di prima coordinata i ha bordo orizzontale pari a $[x(i), x(i+1)]$ per $i = 0, \ldots, N_x + 1$;
- `y`: contiene le coordinate y dei bordi cella: la cella di seconda coordinata j ha bordo orizzontale pari a $[y(j), y(j+1)]$ per $j = 0, \ldots, N_y + 1$;
- `ghost_cells`: la riga i-esima di questo file corrisponde alla i-esima ghost cell. Le prime 2 entrate indicano la label della ghost cell, la terza e la quarta indicano il numero di celle bagnate adiacenti e il numero di punti riflessi che la ghost cell possiede[4]. Seguono 36 entrate a gruppi di 12 che rappresentano nell'ordine
  - le componenti x e y del BP;
  - le componenti x e y del RP;
  - le componenti x e y della normale uscente;
  - le componenti i e j della cella il cui centro coincide col vertice di sud-ovest del IS;
  - le componenti i e j della cella alla quale sono associati BP, RP e la normale;
  - il numero di ghost points presenti in IS;
  - il numero del lato associato.

[4]Di fatto questo numero coincide col numero di celle bagnate adiacenti in modo che ogni cella bagnata abbia il proprio punto riflesso e il proprio punto di bordo.

Qualora la ghost cell presenti meno di tre[5] celle adiacenti bagnate le 12 o 24 entrate mancanti vengono riempite con il valore -1.

Gli `m-files` presenti nella relativa cartella permettono i plot del dominio e di tutte le sue caratteristiche principali.

2.2.3 Pincipali metodi pubblici della classe `grid`

Il listato 2.4 mostra l'utlizzo dei principali metodi pubblici forniti dalla classe `grid`. Alla riga 11 viene costruito l'oggetto G con dominio assegnato per vertici leggendo i file di input di default (vedere il listato 2.2).
Alla riga 12 viene estratto il vettore delle celle bagnate che é di fondamentale importanza per l'applicazione del metodo ai volumi finiti. Si deve infatti ciclare sulle sole celle bagnate e applicare il metodo desiderato senza preoccuparsi della presenza di qualche ghost cell adiacente alla cella bagnata considerata in quel momento.
Il metodo `info` (riga 18) stampa a video le caratteristiche principali del dominio, mentre alla riga 22 viene assegnato il valore alla prima incognita[6] in tutto il dominio (ossia nei centri cella bagnati) attraverso la funzione `sincos`.
Il ciclo alla riga 25 mostra un esempio di iterazione sulle celle bagnate per assegnare i valori della seconda incognita cella per cella. Questo ciclo svolge la stessa funzione del comando

```
G.set_unknown_values(xsquare, 2);
```

I cicli for innestati alle righe 32 e 34, puramente a titolo dimostrativo il cui uso é caldamente sconsigliato, scorrono ogni cella di `G` per assegnare alle sole celle bagnate il valore 4.5 alla terza incognita.
Alla riga 43 viene chiamato il metodo che applica le condizioni al contorno avendo a disposizione tutti i valori delle incognite nelle wet cells. Questo metodo di fatto calcola i valori fittizi da assegnare alle ghost cells per consentire la normale applicazione di un metodo centrato ai volumi finiti.
Alla riga 46 viene estratto il valore della prima incognita in una cella bagnata (la prima del vettore *wet_cells*) mentre alla riga 47 vengono estratti i valori della prima incognita nelle 4 celle adiacenti a tale cella attraverso i parametri *z_E, z_W, z_N, z_S* passati per referenza.
Infine alla riga 50 vengono salvati nel file `./data/unknowns/the_unknowns` i valori delle incognite. La componente (i, j) del file rappresenta il valore della $j-$esima incognita assunto nella $i-$esima cella bagnata la quale label si deduce dalla $i-$esima riga del file `wet_cells`.

[5] Nel caso di 3 celle adiacenti bagnate si cerca ugualmente di costruire le matrici di interpolazione che tuttavia risultano nella maggior parte di questi casi non invertibili. Il costruttore segnala le situazioni critiche con delle stampe a video.

[6] Le incognite sono numerate a partire da 1.

```
#include "grid.hpp"

double sincos(const Point2d<double>& P) {return 1.0 + 0.2*sin(P(0))*cos(P(1));}
double xsquare(const Point2d<double>& P) {return P(0)*P(0);}

int main(int argc, char **argv) {

   using namespace std;
   using namespace IMMERSED_BOUNDARY;

   grid G;
   const vector<label>& wet_cells(G.get_wet_cells());
   const double dx = G.get_dx(), dy = G.get_dy();
   const unsigned int Nx = G.get_Nx(), Ny = G.get_Ny();
   double z, z_E, z_W, z_N, z_S;

   // stampa a video e salvataggio su file informazioni relative al dominio
   G.info();
   G.writeout_domain();

   // impostare i valori alle incognite tramite funzione
   G.set_unknown_values(sincos, 1);

   // impostare i valori alle incognite tramite ciclo sulle wet cells
   for (vector<label>::const_iterator it=wet_cells.begin(); \
      it!=wet_cells.end(); it++)
   {
      G.set_unknown_values(*it, 2, xsquare(G.get_cell_center(*it)));
   }

   // impostare i valori tramite ciclo su tutte le celle (sconsigliato)
   for (unsigned int i=0; i<Nx+2; i++)
   {
      for (unsigned int j=0; j<Ny+2; j++)
      {
         if (G.is_wet(i,j))
            G.set_unknown_values(label(i,j), 3, 4.5);
      }
   }

   // applicazione delle condizioni al contorno
   // ==> calcolo valori incognite nei ghost points
   G.boundary_conditions();

   // estrarre i valori delle incognite
   z = G.get_unknown_value(*wet_cells.begin(), 1);
   G.get_stencil_values(*wet_cells.begin(), 1, z_E, z_W, z_N, z_S);

   // salvare su file i valori delle incognite
   G.writeout_unknowns("./data/unknowns/the_unknowns");

   return 0;
}
```

Listing 2.4: Esempio dell'utilizzo dei principali metodi pubblici della classe `grid`.

```
void set_unknown_values(const label&, const unsigned int& un, const double& uv);
void set_unknown_values(p_fun, const unsigned int& un);

Point2d<double> get_cell_center(const label&) const;
const vector<label>& get_wet_cells() const;
bool get_stencil_values(const label, const unsigned int& un, \
   double& Ev, double& Wv, double& Nv, double& Sv);
double get_unknown_value(const label&, const unsigned int& un);

void writeout_unknowns(const string& file_name = "./data/unknowns/unknowns");

bool is_wet(const unsigned int& i, const unsigned int& j) const;
```

Listing 2.5: Signatures dei principali metodi pubblici della classe `grid`. Il parametro *un* é da intendersi come *unknown number* mentre *uv* come *unknown value*. I parametri *Ev, Wv, Nv, Sv* passati per referenza rappresentano i valori dell'incognita selezionata nelle celle Est, Ovest, Nord e Sud rispetto alla wet cell passata (primo parametro del metodo `get_stencil_values`).

2.3 La classe `unknown`

La classe `unknown` permette di gestire le singole incognite e si interfaccia direttamente con la classe `grid`. In effetti con l'utilizzo della classe `unknown` il compito principale di un oggetto di classe `grid`, una volta costruito il dominio, si riduce essenzialmente alla sola applicazione delle condizioni al contorno, mediante la risoluzione dei sistemi lineari per trovare i valori da assegnare nei ghost points. Il listato 2.6 mostra l'utilizzo di oggetti di classe `unknown`.
Costruita la griglia `G` alla riga 11 viene costruita l'incognita `u` alla riga 12 passando come parametri `G` e il numero dell'incognita (1). Il costruttore della classe `unknown` ha la firma

```
unknown(grid&, const unsigned int& un, const p_comp& p_c = compare);
```

Si osservi che la classe `unknown` non ha un costruttore di default in modo da forzare l'interfacciamento con un oggetto di classe `grid`. Il secondo parametro del costruttore é il numero dell'incognita a cui fa riferimento l'oggetto di classe `unknown`.
Alla riga 23 vengono importati da `G` i valori dell'incognita (la prima in questo caso) in tutto il dominio (wet cells e ghost cells).
Alla riga 27 viene estratto da `u` il valore nella cella bagnata di label `a_wet_label` mentre alla riga 28 vengono estratti i valori di `u` nello stencil di questa cella con la stessa modalità vista per la classe `grid`.
Alla riga 32 si importano i valori nello stencil di `a_wet_label` per referenza tramite la variabile `u_sv_ref`. Questo permette di non copiare i valori delle incognite rendendo il codice più efficente, come vedremo nell'esempio applicativo.
Alla riga 42 si cicla su tutte le wet cells per imporre il valore 1.0 all'incognita `u` mentre alla riga 48 si esportano in `G` i valori di `u`.

L'operatore () può essere usato solo per estrarre il valore dell'incognita in una wet cell. In effetti non ha senso estrarre il valore in una ghost cell senza specificare la wet cell di riferimento. Se una ghost cell confina con più di una wet cell può avere diversi valori dell'incognita a seconda della wet cell alla quale si riferisce.
Nel metodo `get_stencil_values` viene passata la cella bagnata (cella centrale) e quindi é possibile estrarre i valori di tutte le celle adiacenti, siano esse wet o ghost.
Il metodo `export_unknown` copia sulla griglia i valori dell'incognita solo nelle celle bagnate mentre il metodo `import_unknown` importa i valori nelle wet cell e nelle ghost cell.

```cpp
#include "grid.hpp"
#include "unknown.hpp"

double sincos(const Point2d<double>& P) {return 1.0 + 0.2*sin(P(0))*cos(P(1));}

int main(int argc, char **argv) {

   using namespace std;
   using namespace IMMERSED_BOUNDARY;

   grid G;
   unknown u(G,1);
   const vector<label>& WC(G.get_wet_cells()); // wet cells
   label a_wet_label(*WC.begin());
   double u_central, u_E, u_W, u_N, u_S;

   G.set_unknown_values(sincos, 1);

   G.boundary_conditions();

   // importo da G il valore della prima incognita
   // (valori nelle wet cells e nelle ghost cells)
   u.import_unknown();

   // importo valori dell'incognita nella cella centrale
   // e nelle quattro celle adiacenti
   u_central = u(a_wet_label);
   u.get_stencil_values(a_wet_label, u_E, u_W, u_N, u_S);

   // importo PER REFERENZA i valori dell'incognita nella cella centrale
   // e nelle quattro celle adiacenti
   stencil_values u_sv_ref = u.get_stencil_values(a_wet_label);

   // importo solo il valore EST per valore
   u_E = u.get_stencil_value(a_wet_label, unknown_position::E, false);

   // importo solo il valore NORD per referenza con controllo cella bagnata
   double& u_N_ref = u.get_stencil_value(a_wet_label, unknown_position::N,true);


   // assegno il valore 1.0 ad u in tutto il dominio
   for (vector<label>::const_iterator it = WC.begin(); it != WC.end(); it++)
   {
      u(*it) = 1.0;
   }

   // esporto i valori di u in G (prima incognita) nelle sole celle bagnate
   u.export_unknown();
}
```

Listing 2.6: Utilizzo della classe `unknown` ed interfaccia con la classe `grid`.

```
namespace unknown_position
{
enum position{E,W,N,S};
}

double& operator()(const label& WC); // <-- cella bagnata!

void get_stencil_values(const label& WC, \
      double& u_E, double& u_W, double& u_N, double& u_S);

stencil_values get_stencil_values(const label& WC);

double& get_stencil_value(const label& WC, \
      const unknown_position::position& adj_pos, \
      const bool& wet_control = true);


void export_unknown(); // nelle sole celle bagnate
void import_unknown(); // in tutte le celle: wet + ghost
```

Listing 2.7: Dichiarazione della struct `position` e signatures dei principali metodi pubblici della classe `unknown`.

2.4 Esempio di applicazione di un metodo ai volumi finiti

In questa sezione mostriamo come usare le classi `grid` e `unknown` per applicare un metodo ai volumi finiti centrato. Ci riferiamo al caso delle Shallow waters[7] e all'applicazione di un metodo che prevede la ricostruzione polinomiale a pezzi della soluzione in tutto il dominio partendo dalle medie di cella. I flussi numerici dipenderanno dai valori ricostruiti delle incognite alle interfacce delle celle. In formule, partendo dalla soluzione numerica all'istante t^n nei centri cella $\mathbf{u}^n_{j,k}$ con $(j,k) \in WC$ dove WC é l'insieme delle labels delle celle bagnate, si ricostruisce la soluzione in tutto il dominio bagnato[8] nel seguente modo, omettendo per semplicità l'apice n relativo al tempo:

$$\mathbf{u}(x,y) = \sum_{(j,k)\in WC} \chi_{j,k}\mathbf{P}_{j,k}(x,y)$$

dove $\chi_{j,k}$ é l'indicatore della cella $C_{j,k}$ e $\mathbf{P}_{j,k}$ é un vettore di polinomi (un polinomio per ogni incognita) che viene costruito a partire dalle medie di cella al tempo t^n. Supponiamo che la soluzione numerica al tempo t^{n+1} nella cella (j,k) dipenda dalla soluzione nella cella (j,k) al tempo t^n e dai due valori (limite esterno e limite interno) della soluzione ricostruita nei quattro punti medi dei lati della cella (j,k), come illustrato in figura 2.1. I polinomi $\mathbf{P}_{j,k}$ dipendono, nel caso più semplice, dal valore $\mathbf{u}_{j,k}$ e dai quattro valori adiacenti $\mathbf{u}_{j\pm1,k\pm1}$.
Il listato 2.8 mostra l'applicazione del metodo ai volumi finiti. Dopo aver definito tutte le variabili necessarie si salvano le caratteristiche del dominio computazionale (riga 28) e si applicano le condizioni iniziali direttamente alla griglia `G` (riga 30).

[7] Sebbene l'esempio sia facilmente estendibile al caso generale.
[8] Ossia il dominio formato dalle celle con centro bagnato.

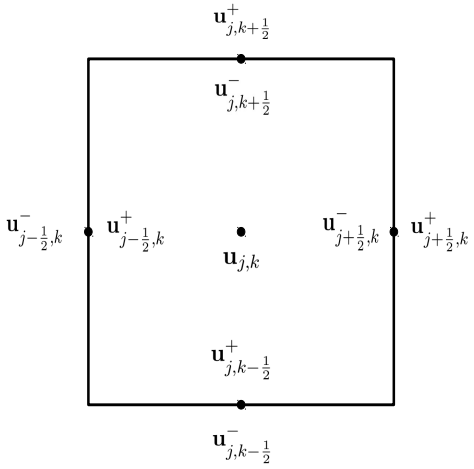

Figura 2.1: Dominio di dipendenza della soluzione numerica $\mathbf{u}^{n+1}_{j,k}$. Tutte le quantità in figura sono calcolate al tempo t^n.

Segue poi il ciclo temporale che ad ogni passo calcola la soluzione al tempo t^{n+1} partendo dalla soluzione al tempo t^n:

- salvataggio delle incognite della soluzione al tempo t^n nei centri cella bagnati (celle computazionali);
- imposizione delle condizioni al contorno per i valori di centro cella, ossia calcolo dei valori di centro cella per i ghost points;
- ricostruzione polinomiale a pezzi della soluzione al tempo t^n e calcolo dei valori di interfaccia interna[9] per ogni cella bagnata;
- imposizione delle condizioni al contorno per i valori di interfaccia interna, ossia calcolo dei valori di interfaccia interna per le ghost cells;
- calcolo del passo temporale;
- calcolo della soluzione al tempo t^{n+1}.

L'imposizione delle condizioni al contorno per il calcolo dei valori di interfaccia interna nelle ghost cells avviene esportando uno alla volta i quattro valori di interfaccia sulla griglia `G`, cioè traslando i valori dell'interfaccia selezionata al centro delle celle bagnate e applicando le condizioni al contorno come se il valore di interfaccia fosse assunto al centro della cella.

[9]Data la cella (j,k) e l'incognita $\mathbf{u}$ i valori di interfaccia interna sono le quantità $\mathbf{u}^-_{j+\frac{1}{2},k}$, $\mathbf{u}^+_{j-\frac{1}{2},k}$, $\mathbf{u}^-_{j,k+\frac{1}{2}}$ e $\mathbf{u}^+_{j,k-\frac{1}{2}}$.

```cpp
#include "finite_volume.hpp"

int main(int argc, char **argv) {

    using namespace std;
    using namespace IMMERSED_BOUNDARY;
    using namespace FINITE_VOLUME;

    grid G;
    const vector<label>& WC(G.get_wet_cells());
    unsigned int n_iteration = 0;
    const double CFL = 0.9;
    const double dx = G.get_dx(), dy = G.get_dy();
    const double t_final = 1.0;
    double t = 0.0;
    double dt;
    // incognite: valori centrali e valori alle interfacce ->
    // -> limiti dall'interno della cella
    cell_unknown z(G,1);
    cell_unknown Qx(G,2);
    cell_unknown Qy(G,3);

    // valori delle incognite (a centro cella) nel nuovo istante temporale
    unknown new_z(G,1);
    unknown new_Qx(G,2);
    unknown new_Qy(G,3);
    ///////////////
    G.writeout_domain(false); // salvataggio in ./data/domain
    // Imposizione delle condizioni iniziali in G
    initial_conditions(G);

    while (t < t_final)
    {
        // (1) Salvataggio incognite (medie di cella nelle wet cells)
        G.writeout_unknowns("./data/unknowns/frame_" + int2string(n_iteration));
        n_iteration++;
        // (2) Imposizione delle condizioni al bordo (medie di cella)
        // necessarie alla fase di ricostruzione
        G.boundary_conditions();
        // estrazione delle incognite nelle wet e ghost cells
        z.central.import_unknown();
        Qx.central.import_unknown();
        Qy.central.import_unknown();
        // (3) Ricostruzione lineare a pezzi delle variabili: calcolo dei valori
        // alle 4 interfacce (limiti interni) per ogni wet cell
        reconstruction(G, z, Qx, Qy, WC);
        // (4) Imposizione delle condizioni al contorno ai valori di interfaccia
        // necessarie all'applicazione del metodo centrato
        interfaces_boundary_conditions(G, z, Qx, Qy);
        // (5) calcolo di dt
        dt = time_step(z, Qx, Qy, WC, dx, dy, CFL);
        t += dt;
        cout<< "dt=" <<dt <<"; t=" <<t <<"; iterazione: " <<n_iteration <<endl;
        // (6) calcolo incognite al tempo successivo
        evolution(z, Qx, Qy, new_z, new_Qx, new_Qy, WC, dx, dy, dt);
    }
}
```

Listing 2.8: Uso delle classi `grid` e `unkwnown` per l'applicazione di un metodo ai volumi finiti.

```
struct cell_unknown // cell_unknown u
{
   // label wc(j,k) --> una cella bagnata
   unknown central; // u.central(j,k) --> u(j,k)
   unknown E; // u.E(j,k) --> u(j+1/2, k)^-
   unknown W; // u.W(j,k) --> u(j-1/2, k)^+
   unknown N; // u.N(j,k) --> u(j, k+1/2)^-
   unknown S; // u.S(j,k) --> u(j, k-1/2)^+
};

struct triplet
{
   double& z;
   double& Qx;
   double& Qy;
};

struct interfaces_values
{
   triplet E;
   triplet W;
   triplet N;
   triplet S;
};
```

Listing 2.9: Strutture nel namespace `FINITE_VOLUME`.

2.4.1 Analisi del codice

Il listato 2.8 definisce le incognite `z`, `Qx` e `Qy` di tipo `cell_unknown`. Si tratta di una semplice struttura nel namespace `FINITE_VOLUME` per trattare i valori a centro cella e nelle quattro interfacce interne. Questo namespace definisce anche le strutture `triplet` e `interface_values` per il trattamento semplice e intuitivo dei valori delle incognite a centro cella e alle interfacce una volta definita la cella di riferimento. Ognuna di queste strutture é provvista di uno o più costruttori non riportati nel listato 2.9. Alla riga 28 vengono salvate le caratteristiche del dominio come visto nella sezione 2.2.2 mentre alla riga 30 vengono applicate le condizioni iniziali (vedere il sorgente `example/set_initial_conditions.cpp`).
Alla riga 32 inizia il ciclo temporale per il calcolo della soluzione numerica fino al tempo `t_final`. Alla riga 35 vengono salvate le incognite al tempo t^n con il nome `frame_n` dove n é il numero dell'iterazione in modo da ordinare i file di output per la visualizzazione. La funzione `int2string` é contenuta nel namespace `IMMERSED_BOUNDARY`.
Dopo aver applicato le condizioni al contorno ed importato i valori delle incognite (riga 39 e righe 41, 42, 43) vengono ricostruite polinomialmente le incognite.
Alla riga 49 vengono applicate le condizioni al contorno per i valori di interfaccia necessari all'evoluzione temporale che avviene alla riga 55.

Ricostruzione

Alla riga 13 del listato 2.10 vengono estratti per referenza i valori di centro cella dell'incognita z nel nodo centrale e nelle quattro celle adiacenti, ossia i valori $z_{j,k}$, $z_{j\pm1,k}$ e $z_{j,k\pm1}$ fissata una certa wet cell (j,k).
Il cuore della funzione `reconstruction` consiste nell'impiego alla riga 15 della funzione `cell_reconstruction` che noti i valori delle medie di z nello stencil calcola e assegna, attraverso i parametri passati per referenza, i valori di interfaccia interna. Il listato 2.11 mostra l'implementazione di una ricostruzione lineare a pezzi dove le derivate parziali ven-

```cpp
#include "finite_volume.hpp"

void FINITE_VOLUME::reconstruction(grid& g, cell_unknown& z, cell_unknown& Qx, \
      cell_unknown& Qy, const std::vector<IMMERSED_BOUNDARY::label>& WC)
{

   using namespace IMMERSED_BOUNDARY;
   using namespace FINITE_VOLUME;

   for (std::vector<label>::const_iterator it=WC.begin(); it!=WC.end(); it++)
   {
      // import delle medie di cella tramite referenze;
      const stencil_values z_sv = z.central.get_stencil_values(*it);
      // calcolo valori interfaccia interna;
      cell_reconstruction(z_sv, z.E(*it), z.W(*it), z.N(*it), z.S(*it));

      const stencil_values Qx_sv = Qx.central.get_stencil_values(*it);
      cell_reconstruction(Qx_sv, Qx.E(*it), Qx.W(*it), Qx.N(*it), Qx.S(*it));

      const stencil_values Qy_sv = Qy.central.get_stencil_values(*it);
      cell_reconstruction(Qy_sv, Qy.E(*it), Qy.W(*it), Qy.N(*it), Qy.S(*it));
   }
}
```

Listing 2.10: Funzione `reconstruction` nel namespace `FINITE_VOLUME`.

```cpp
#include "finite_volume.hpp"

void FINITE_VOLUME::cell_reconstruction( \
      const IMMERSED_BOUNDARY::stencil_values& u, \
      double& u_Em, double& u_Wp, double& u_Nm, double& u_Sp)
{

   using namespace FINITE_VOLUME;

   // u.E --> u(j+1,k); u.S --> u(j,k-1)
   double u_x = minmod(u.E-u.central, u.central-u.W, 0.5*(u.E-u.W));
   double u_y = minmod(u.N-u.central, u.central-u.S, 0.5*(u.N-u.S));


   u_Em = u.central + 0.5*u_x; // u_(j+1/2,k)^-
   u_Wp = u.central - 0.5*u_x; // u_(j-1/2,k)^+
   u_Nm = u.central + 0.5*u_y; // u_(j,k+1/2)^-
   u_Sp = u.central - 0.5*u_y; // u_(j,k-1/2)^+
}
```

Listing 2.11: Funzione `cell_reconstruction` nel namespace `FINITE_VOLUME`.

gono stimate tramite la funzione *minmod* alla quale vengono passate le approssimazioni alle differenze finite in avanti, all'indietro e centrata delle derivate (righe 11 e 12). La funzione minmod é così definita:

$$minmod(x,y,z) = \begin{cases} \min(x,y,z) & \text{se} \quad x,y,z > 0; \\ \max(x,y,z) & \text{se} \quad x,y,z < 0; \\ 0 & \text{altrimenti.} \end{cases}$$

Condizioni al bordo per i valori di interfaccia

Supponiamo di voler calcolare la soluzione numerica al nuovo istante temporale in una cella (j^*, k^*) e che la cella (j^*-1, k^*) sia una ghost cell, mentre le altre tre celle adiacenti siano tutte bagnate. Per applicare il metodo dei volumi finiti descritto mancano i valori delle incognite nell'interfaccia esterna dal lato ovest, ossia le quantitá $\mathbf{u}^-_{j-\frac{1}{2},k}$. Per recuperare questi valori bisogna applicare le condizioni al contorno tramite la griglia G. Poichè i valori $\mathbf{u}^-_{j+\frac{1}{2},k}$ sono disponibili in tutte le celle bagnate, tramite la ricostruzione fatta precedentemente che utilizzava i valori centrali nei ghost points, si é deciso di operare come segue: si esportano sulla griglia i valori $\mathbf{u}^-_{j+\frac{1}{2},k}$ come se fossero assunti al centro delle celle bagnate e si applicano le condizioni al bordo per recuperare i valori nei ghost points. In questo modo diventa disponibile il valore $\mathbf{u}^-_{j^*-1+\frac{1}{2},k^*}$ necessario all'applicazione del metodo. Il codice che implementa le condizioni al bordo per le interfacce si trova nel sorgente `example/interfaces_boundary_conditions.cpp`.

Calcolo del passo temporale

La funzione `time_step` calcola il passo temporale come

$$\Delta t = \mathrm{CFL} \min\left(\frac{\Delta x}{\Lambda_x}, \frac{\Delta y}{\Lambda_y}\right)$$

dove Λ_x e Λ_y sono i valori massimi assunti dagli autovalori delle matrici Jacobiane $[\mathbf{f}(\mathbf{u})]_x$ e $[\mathbf{g}(\mathbf{u})]_y$. Per maggiori dettagli far riferimento alla teoria dei metodi ai volumi finiti.

Evoluzione temporale

Una volta ricostruita polinomialmente la soluzione e tutti i valori delle incognite nei ghost points la funzione `evolution` calcola il valore delle incognite nel nuovo istante temporale. Sostanzialmente i metodi ai volumi finiti centrati si differenziano per il tipo della ricostruzione della soluzione a partire dalle medie di cella e per lo schema col quale si calcolano i valori $\mathbf{u}^{n+1}_{j,k}$ a partire dai valori $\mathbf{u}^n_{j,k}$. Il codice proposto, pensato per il sistema delle Shallow Waters ma facilmente generalizzabile[10], é quindi in grado di risolvere diversi metodi centrati semplicemente agendo sulle funzioni `cell_reconstruction` e `cell_evolution`. Il listato 2.12 implementa l'evoluzione temporale una volta ricostruita la soluzione e calcolato Δt. Alla riga 16 vengono estrapolati, tramite referenze costanti, i valori centrali delle incognite nella cella corrente `*it` mentre alla riga 26 vengono estrapolati i valori di interfaccia esterna dal lato EST. L'ultimo parametro stabilisce la posizione rispetto alla celle corrente `*it` della cella dalla quale si intende estrapolarne i valori. Se (j,k) é la cella corrente alla riga 26 si estrapolano quindi i valori $\mathbf{u}^+_{(j+1)-\frac{1}{2},k}$ nella variabile `u_Ep`.
Il cuore dell'evoluzione temporale risiede alla riga 32.
La signature della funzione `cell_evolution` é

[10] A meno della gestione delle condizioni al bordo.

```
#include "finite_volume.hpp"

void FINITE_VOLUME::evolution(cell_unknown& z, cell_unknown& Qx, \
   cell_unknown& Qy, unknown& new_z, unknown& new_Qx, unknown& new_Qy, \
   const std::vector<IMMERSED_BOUNDARY::label>& WC, \
   const double& dx, const double& dy, const double& dt)
{

   using namespace IMMERSED_BOUNDARY;
   using namespace FINITE_VOLUME;
   // u_c: medie di cella u(j,k)
   // u_Em = u(j+1/2,k)^-, ... , u_Sp = u(j,k-1/2)^+
   for (std::vector<label>::const_iterator it=WC.begin(); it!=WC.end(); it++)
   {
      // medie di cella
      const triplet u_c(z.central, Qx.central, Qy.central, *it);

      //interfacce EST, OVEST, NORD, SUD interne (u_Nm --> u(j,k+1/2)^-)
      const triplet u_Em(z.E, Qx.E, Qy.E, *it);
      const triplet u_Wp(z.W, Qx.W, Qy.W, *it);
      const triplet u_Nm(z.N, Qx.N, Qy.N, *it);
      const triplet u_Sp(z.S, Qx.S, Qy.S, *it);
      const interfaces_values intern_interf(u_Em, u_Wp, u_Nm, u_Sp);

      // interfacce EST, OVEST, NORD, SUD esterne (u_Ep --> u(j+1/2,k)^+)
      const triplet u_Ep(z.W, Qx.W, Qy.W, *it, unknown_position::E);
      const triplet u_Wm(z.E, Qx.E, Qy.E, *it, unknown_position::W);
      const triplet u_Np(z.S, Qx.S, Qy.S, *it, unknown_position::N);
      const triplet u_Sm(z.N, Qx.N, Qy.N, *it, unknown_position::S);
      const interfaces_values extern_interf(u_Ep, u_Wm, u_Np, u_Sm);
      // --> EVOLUZIONE <--
      cell_evolution(new_z(*it), new_Qx(*it), new_Qy(*it), u_c, \
         intern_interf, extern_interf, dx, dy, dt);
   }
   new_z.export_unknown();
   new_Qx.export_unknown();
   new_Qy.export_unknown();
}
```

Listing 2.12: Funzione `evolution` nel namespace `FINITE_VOLUME`.

```
void cell_evolution(double& new_z, double& new_Qx, double& new_Qy, \
     const triplet& u_c, \
     const interfaces_values& minus_interf, \
     const interfaces_values& plus_interf, \
     const double& dx, const double& dy, const double& dt);
```

Questa funzione applica il metodo ai volumi finiti centrato

$$\mathbf{u}_{j,k}^{n+1} = \mathbf{\Phi}\left(\mathbf{u}_{j,k}^{n}, \mathbf{u}_{j+\frac{1}{2},k}^{\pm}, \mathbf{u}_{j-\frac{1}{2},k}^{\pm}, \mathbf{u}_{j,k+\frac{1}{2}}^{\pm}, \mathbf{u}_{j,k-\frac{1}{2}}^{\pm}\right)$$

dove $\mathbf{\Phi}$ dipende dai flussi numerici e dal tipo di evoluzione temporale che si vuole impiegare.

2.5 `m-files` per le visualizzazioni con `Matlab`

La cartella `m-files` contiene utili codici per la visualizzazione dei risultati con `Matlab`. Le funzioni più importanti sono:

- `plot_domain.m`: esegue il plot delle celle quadrate, del dominio reale e del dominio computazionale. Permette di evidenziare un particolare lato, una cella o una ghost cell;
- `plot_grid_edges.m`: esegue il plot della griglia cartesiana e del dominio computazionale;
- `plot_unknown.m`: esegue il plot di una delle incognite indicate mediante supericie e/o insiemi di livello.

Per maggiori dettagli riguardo i parametri d'ingresso e uscita consultare l'help delle funzioni.

Capitolo 3

Estensione del codice

In questo capitolo ci occupiamo delle modifiche da apportare al codice per l'applicazione dei metodi ai volumi centrati a diversi sistemi di equazioni. Le modifiche sostanziali riguardano la costruzione delle matrici di interpolazione poichè, come visto nel precedente capitolo, le uniche modifiche da fare per risolvere numericamente un certo sistema di equazioni con diversi metodi centrati riguardano solamente le funzioni `cell_reconstruction` e `cell_evolution`.

3.1 Costruzione delle matrici di interpolazione

Le matrici di interpolazione vengono assemblate dal costruttore della classe `grid` mediante il metodo privato `build_interpolation_matrices`. Come visto nel capitolo 1 le matrici sono associate ad un certo quadrato di interpolazione ed il tipo di matrice da assemblare dipende dalle caratteristiche dei vertici di interpolazione. I casi da considerare (*all_wet*, *GPs* e *dry_no_GP*) sono indipendenti dal tipo di sistema e dal numero di equazioni che lo compongono. Il namespace `IMMERSED_BOUNDARY` definisce utili typedef per la gestione delle matrici, che avviene tramite l'impiego del software `Eigen`[1] (vedere il listato 3.1).
Le matrici di interpolazione vengono salvate con la loro decomposizione LU poichè l'applicazione delle condizioni al bordo ai vari istanti temporali si differenzia solo per il termine noto mentre le matrici non dipendono dal tempo. Come visto nel capitolo 1 nel caso *all_wet* si deve salvare solo una matrice di interpolazione di dimensioni 4x4. Questo caso non solo é indipendente dalle condizioni al bordo che si vogliono implementare ma anche dal sistema di equazioni da risolvere, mentre gli altri casi dipendono da entrambi. Nel caso `GPs`, ad esempio, il sistema delle Shallow Waters necessita di una matrice 4x4 per il calcolo dell'altezza e una matrice 8x8 per il calcolo delle portate. I tipi delle matrici da salvare si differenziano per il tipo di quadrato di interpolazione. Nel caso delle Shallow Waters si hanno i tipi definiti alle righe 15, 19 e 22 del listato 3.1. Nel caso *dry_no_GP* é necessario salvare la posizione del vertice dry non ghost[2].
Le matrici sono associate alla label della cella che contiene il vertice di sud-ovest del quadrato di interpolazione e vengono perciò gestite dalla classe `grid` mediante i contenitori `map` della standard library.
Le singole matrici vengono costruite attraverso le funzioni

- `build_all_wet_4x4_matrix`
- `build_GPs_8x8_matrix`
- `build_GPs_4x4_matrix`

[1] *http://eigen.tuxfamily.org/index.php?title=Main_Page*
[2] Ricordiamo che la numerazione é 1 per il vertice SW, 2 per quello di SE, 3 per il NE e 4 per il NW.

```
typedef Eigen::Matrix<double, 3, 3> M3;
typedef Eigen::Matrix<double, 4, 4> M4;
typedef Eigen::Matrix<double, 6, 6> M6;
typedef Eigen::Matrix<double, 8, 8> M8;

typedef Eigen::Matrix<double, 3, 1> V3;
typedef Eigen::Matrix<double, 4, 1> V4;
typedef Eigen::Matrix<double, 6, 1> V6;
typedef Eigen::Matrix<double, 8, 1> V8;

typedef std::pair<M8,M4> M8M4;
typedef std::pair<M6,M3> M6M3;

typedef Eigen::FullPivLU<M3> M3_LU;
typedef Eigen::FullPivLU<M4> M4_LU; // <-- case all_wet matrices type
typedef Eigen::FullPivLU<M6> M6_LU;
typedef Eigen::FullPivLU<M8> M8_LU;

typedef std::pair<M8_LU,M4_LU> M8M4_LU; // <-- case GPs matrices type
typedef std::pair<M6_LU,M3_LU> M6M3_LU;

struct M6M3_LU_non_cc // <-- case dry_no_GP matrices type
{
   M6M3_LU matrices;
   unsigned int non_cc; // non considered corner (dry and non ghost corner)
};
```

Listing 3.1: Typedef nel namespace `IMMERSED_BOUNDARY` per la gestione delle matrici .

```
std::map<label, M4_LU, p_comp, \
Eigen::aligned_allocator<std::pair<const label, M4_LU> > > all_wet_matrices;

std::map<label, M8M4_LU, p_comp, \
Eigen::aligned_allocator<std::pair<const label, M8M4_LU> > > GPs_Matrices;

std::map<label, M6M3_LU_non_cc, p_comp, \
Eigen::aligned_allocator<std::pair<const label, M6M3_LU_non_cc> > > noGPs_Matrices;
```

Listing 3.2: Attributi privati della classe `grid` per la gestione delle matrici di interpolazione. É stato omesso lo scope identifier `IMMERSED_BOUNDARY`.

- `build_noGPs_6x6_matrix`
- `build_noGPs_3x3_matrix`

implementate nel source file `grid_build_interpolation_matrices.cpp`.
Le modifiche da fare riguardano quindi la definizione dei tipi delle matrici nei 3 casi e nella costruzione effettiva di tali matrici. Nel caso in cui le matrici dipendano anche dal tipo di condizioni al bordo bisogna tener conto della condizione associata al lato contenente il boundary point.

3.1.1 Applicazione delle condizioni al contorno

Una volta assemblate le matrici si calcolano i termini noti che permettono di determinare i valori delle incognite nei punti riflessi attraverso la risoluzione dei sistemi lineari. Nel caso delle Shallow waters i termini noti non dipendono dalle condizioni al contorno nel caso *all_wet*, mentre vi dipendono negli altri due casi. I calcoli necessari sono implementati dal membro privato `calculate_RP_quantities`. Una volta noti i valori nei punti riflessi il membro privato `calculate_GP_quantities` trova finalmente i valori delle incognite nei ghost points.
La classe `grid` é provvista dell'attributo

```
std::vector<unsigned int> type_of_boundary_conditions;
```

per la gestione del tipo di condizioni al contorno (identificate da *unsigned int*) e dei metodi `set_BC_type` per l'impostazione del tipo di condizioni al contorno. Il vettore ha dimensione pari al numero di lati del dominio computazionale e la sua componente i esprime il tipo di condizione al contorno da applicare al lato $i+1$, con $i = 0, \ldots, N_e - 1$ dove N_e é il numero di lati.

Appendice A

Tecniche di programmazione usate

Il progetto fa uso estensivo dei contenitori generici `vector` e `map` della standard library. La classe `grid` impiega la tecnica del contenimento: possiede infatti l'attributo privato `ccb` (cut cell boundary) che é un oggetto di classe `boundary` pensato per la gestione e costruzione del dominio computazionale. La classe `grid` utilizza ampiamente anche le classi `cell` e `ghost_cell`, la quale eredita pubblicamente dalla classe `cell`. La gestione dei punti in due dimensioni avviene per mezzo della classe template `Point2d<T>` dove T=*double* per rappresentare i punti del piano e T=*unsigned int* per rappresentare le etichette di cella.
Il parsing dei dati, seppur elementare, utilizza `GetPot` mentre le matrici vengono gestite grazie al software `Eigen`.