1 Implementazione del codice in C++

Per la risoluzione di leggi di conservazione non lineari 2D su griglie poligonali è stata sviluppata la libreria *ConservationLaw2D*, in gran parte scritta in C++.
La struttura del codice si organizza essenzialmente su tre livelli:

1. La gestione della mesh;
2. La descrizione del modello;
3. Il solutore a Volumi Finiti.

Se, per esempio, volessimo risolvere un particolare problema sulle equazioni di Eulero (come i casi test presentati nelle prossime sezioni), non si deve fare altro che leggere la mesh, definire il modello, condizioni al contorno, iniziali ed eventuali parametri, e passare il tutto al solutore. La soluzione all'istante di tempo desiderato potrà poi essere visualizzata per esempio in MatLab attraverso funzioni appositamente scritte.

1.1 Gestione della mesh

1.1.1 La struttura dati

La parte più sostanziosa e certamente meno banale è quella riguardante la gestione della mesh. Il requisito essenziale nel nostro caso riguarda la possibilità di effettuare particolari "query" di adiacenza tra i vari elementi: primo tra tutti, dato un elemento K, restituire gli elementi ad esso adiacenti, ossia che condividono con esso un lato. Questo tipo di problema può essere risolto in svariati modi, ma ognuno è un compromesso tra utilizzo di memoria ed efficienza.
La struttura dati che abbiamo deciso di adottare è basata sulla tecnica *half-edge*, dove ogni lato è memorizzato come coppia di lati gemelli orientati però in direzioni opposte. In particolare, ogni elemento è descritto da una successione in senso antiorario di half-edge, ognuno dei quali ha un puntatore al suo successivo, al vertice di origine, all'elemento alla sua sinistra (ricordiamo che il segmento è orientato) e al suo lato gemello (se esiste), appartenente al triangolo adiacente a quel lato. Una mesh di questo tipo è detta *edge-based*[1].
La scelta di utilizzare questa struttura dati è nata dalla necessità di trovare qualcosa che non fosse troppo oneroso in termini di memoria (per esempio memorizzando tutte le adiacente e incidenze tra gli elementi), che risultasse sufficientemente efficiente e che fosse abbastanza robusta nell'eventualità di una futura estensione a tecniche di raffinamento e deraffinamento, le quali eliminando triangoli possono compromettere la struttura delle adiacenze.
La seconda scelta progettuale di rilevanza riguarda invece la possibilità di utilizzare mesh composte da poligoni di un numero di lati non necessariamente identico e soprattutto non necessariamente triangolari: i Volumi Finiti infatti si applicano perfettamente a prescindere dal numero di lati del volume di controllo. La struttura dati utilizzata ha permesso di implementare tale scelta senza grossi sforzi aggiuntivi.
Infine una terza importante caratteristica è la possibilità di estendere le classi che descrivono vertice, lato e poligono a proprio piacimento, senza influenzare però la struttura dati. La mesh può

[1] Le mesh basate su half-edge sono spesso utilizzate nella Computer Grafica perché permettono anche di descrivere molti tipi di varietà. Non è però l'unica tecnica esistente, anzi ve ne sono molte altre (winged-edge, quad-edge).

quindi essere puramente topologica (essenzialmente un grafo), oppure avere anche una struttura geometrica o, come nel nostro caso, una struttura di spazio funzionale, dove ogni poligono descrive anche una funzione costante (e dunque la mesh descrive una funzione costante a tratti). In questo ambito si è fatto ricorso ad un uso moderato di *templates*.

1.1.2 Breve introduzione alle classi principali

Le due parti principali della mesh sono il *Kernel* e i *Circulators*. La prima è la struttura dati vera e propria, la seconda invece permette di risolvere le adiacenze, "circolando" ad esempio attraverso i poligoni adiacenti ad uno dato.
Il *Kernel* è composto da 5 classi:

- `BaseKernel`, che non è nient'altro che un contenitore di tipi che saranno comuni a tutti;
- `BaseVertex`, la classe basilare del vertice, che contiene le coordinate e un puntatore ad un halfedge che parte da esso[2];
- `BaseHEdge`, la classe che descrive l'halfedge, con puntatori al vertice di partenza, al lato successivo, a quello gemello e anche un puntatore al poligono alla sua sinistra. Vi sono poi metodi per accedere ai due vertici e ai due poligoni adiacenti (sempre se quello alla destra esiste);
- `BasePolygon`, la classe che descrive il poligono, contiene al suo interno un puntatore ad uno dei suoi lati;
- `BasePolygonalMesh`, ossia la mesh vera e propria che mette a disposizione iteratori sui suoi elementi nonché metodi per aggiungere nuovi poligoni e vertici.

Inoltre sia i vertici che i poligoni hanno un attributo `color`, come fosse un marker che ci permette di distinguerli. Ad esempio questo ci permette di assegnare un certo valore di soluzione solo ad alcuni triangoli che abbiamo precedentemente marcato, e stessa cosa per i lati (utile nell'assegnazione delle condizioni iniziali e di bordo).
Tutte queste classi hanno il prefisso `Base` perché devono essere necessariamente estese, eventualmente da classi vuote.
Per quanto riguarda le adiacenze, abbiamo essenzialmente 6 tipi di "query" che si possono effettuare: 3 per i vertici, ossia la lista dei poligoni, lati e vertici attorno al vertice dato, e 3 per i poligoni.
Tutte queste 6 tipi di adiacenze possono essere risolte dalla specializzazione dell'unica classe `CirculatorTS`. L'idea che sta alla base del *circolatore* (e non *iteratore*) è che il contenitore sottostante non ha un inizio e una fine, bensì si chiude ad anello. La struttura si presta poi bene a queste operazioni di circolazione: prendiamo come esempio i poligoni adiacenti ad uno dato:

Init Parto da un half-edge qualsiasi del poligono;

Increment Dall'half-edge considero quello successivo e lo imposto come corrente;

Reading Restituisco il poligono dell'half-edge gemello.

[2] in realtà è una lista di puntatori, perché alcuni vertici possono essere degeneri, come ad esempio nel caso di due triangoli con in comune un solo vertice.

Vi sono poi delle modifiche che bisogna apportare per gestire poligoni di bordo (dove l'operazione *Reading* non si può effettuare), particolarmente complesse in alcuni casi.

1.1.3 Specializzazione per Volumi Finiti

Le classi del *Kernel*, come accennato, devono essere necessariamente estese per essere utilizzabili. Nel nostro caso sono state specializzate per i Volumi Finiti, dove semplicemente in ogni poligono è stato aggiunto un attributo per la soluzione del problema. Questa scelta di incapsulare i dati e di non creare una classe apposita per le funzioni permette di avere tempi rapidi di accesso ai dati.
Inoltre sono stati aggiunti dei metodi per calcolare una volta per tutte quantità geometriche importanti (aree, lunghezze, baricentri), in modo da velocizzare il codice a costo di un utilizzo maggiore di memoria.

1.2 Descrizione del modello

La seconda parte del codice di *ConservationLaw2D* riguarda la descrizione di un *modello*.
La classe `Eulero` è appunto un modello. In questa classe sono presenti metodi per il passaggio dalle variabili primitive a quelle conservate (e viceversa), per il calcolo del flusso numerico e degli autovalori, e così via.
Il *solutore* poi sarà specializzato ad un particolare modello, e le chiamate ai metodi citati saranno del tutto trasparenti. In questo modo è molto semplice scrivere un codice che gestisca modelli come le *Shallow Waters*.
Il modello però deve mettere a disposizione (in classi separate) anche dei flussi numerici, in particolare per solutori approssimati del problema di Riemann. Infatti generalmente i flussi numerici (escluso Lax-Friedrichs, implementato invece nel solutore) devono essere scritti esplicitamente per il modello in questione.

1.3 Il solutore a Volumi Finiti

Il solutore vero e proprio non fa nient'altro che prendere in ingresso mesh e modello ed implementare lo schema (**??**). In particolare per prima cosa deve essere inizializzato (dove ad esempio calcola il dato iniziale), e poi si possono effettuare passi temporali (passo da $\mathbf{U}^n$ a $\mathbf{U}^{n+1}$). Tutto il necessario per il passo temporale è messo a disposizione dal modello e dai circolatori della mesh. La soluzione ad un dato istante può poi essere salvata su file nella cartella `data`.

1.4 Un esempio pratico

Vediamo un esempio di codice per risolvere il problema Sod con metodo di Roe, descritto più avanti.

```
1 #include <models/eulero/eulero.hpp>
2 #include <models/eulero/fluxes/godunovROE.hpp>
3 #include <solvers/finitevolume.hpp>
4 #include <mesh/io/meshreader.hpp>
```

Nell'intestazione viene caricato il modello di Eulero, il flusso di Godunov con approssimazione alla Roe, il solutore a Volumi Finiti ed infine una funzione per leggere la Mesh.

```
5  using namespace ConservationLaw2D;
6
7  typedef double                                          real_t;
8  typedef Model::Eulero<real_t>                           myModel;
9  typedef NumericalFlux::GodunovRoe<myModel>              myNumFlux;
10 typedef Solver::FiniteVolume<myModel,myNumFlux>         mySolver;
11 typedef mySolver::FVMesh                                myMesh;
12
13 typedef myModel::SolType                                SolType;
```

In questa sezione vengono definiti i tipi e viene costruito il solutore. In particolare viene definito il modello di Eulero (con dati di tipo `double`), il flusso numerico, il solutore e la mesh. Infine il tipo di dato `SolType` è un vettore m-dimensionale, dove m è il numero di equazioni del sistema (per Eulero $m = 4$).

```
14 inline SolType init( size_t color, real_t x, real_t y ) {
15    SolType sol;
16    sol[0] = (x < 0) ? 1.0 : 0.125;
17    sol[1] = (x < 0) ? 0.75 : 0.0;
18    sol[2] = 0.0;
19    sol[3] = (x < 0) ? 1.0 : 0.1;
20    return sol;
21 }
22
23 inline SolType bc( const SolType& wl, size_t color, real_t x, real_t y, real_t nx,
       real_t ny, real_t t ) {
24    return wl;
25 }
```

Le condizioni iniziali e quelle al bordo sono definite come due funzioni. Il solutore al momento dell'inizializzazione e del passo temporale si preoccuperà di effettuare chiamate a queste due funzioni. Osserviamo che in questo caso usiamo condizioni al bordo assorbenti ovunque, cioè imponiamo che nell'elemento fantasma ci sia il dato a sinistra del lato.

```
26 int main(int argc, char **argv) {
27    // Avvertimento
28    if ( argc < 2 ) {
29       cout << "Usage: " << argv[0] << " meshfile.msh" << endl;
30       exit(EXIT_SUCCESS);
31    }
32    // Definisco il modello (con gamma=1.4)
33    myModel model(1.4);
34    // Definisco la mesh
35    myMesh mesh;
36    // Leggo la mesh
37    Mesh::IO::MeshReader(mesh, argv[1]);
38    // Definisco il solutore per il mio modello
39    mySolver solver(model, mesh);
40    // Inizializzo alcuni parametri
41    solver.setCFLmax(0.3);
42    solver.setIC(init);
43    solver.setBC(bc);
44    // Inizializzo il solutore
45    solver.init();
46    // Passi temporali
47    for (int i = 0; i <= 1000; ++i) {
48       solver.timestep();
49       solver.framegrab(i);
50    }
51    return 0;
52 }
```

Veniamo ora al programma principale. Il workflow è piuttosto semplice:

1. Creiamo la classe per il modello, assegnando in questo caso $\gamma = 1.4$;

2. Definisco poi la mesh che leggo dal file assegnato;
3. Creiamo il solutore passando referenze della mesh e del modello;
4. Imposto CFL desiderato, condizioni iniziali e condizioni al bordo e inizializzo il solutore;
5. Faccio 1000 (in questo caso) passi temporali, stampando la soluzione per ognuno.