ADATTIVITÀ DI GRIGLIA

ALFONSO FASCÌ

1. Abstract

In questo lavoro è presentata l'estensione di un codice implementato in C++, per il calcolo di soluzioni approssimate con elementi finiti lineari (Metodo di Galerkin Elementi Finiti) del problema di Poisson su un dominio bidimensionale poligonale convesso, con dato al bordo di Dirichlet omogeneo. L'estensione consiste nell'implementazione di un algoritmo di adattività di griglia.

2. Risultati Teorici

Si consideri il seguente problema:

$$\begin{cases} \Delta u = f & \text{su } \Omega \\ u = 0 & \text{su } \partial\Omega \end{cases} \tag{1}$$

in cui $\Omega \subset \mathbb{R}^2$ poligonale convesso limitato e $f \in L^2(\Omega)$. Sia u_h un'approssimazione di u a elementi finiti lineari ottenuta tramite il metodo di Galerkin ed associata ad un'opportuna triangolazione conforme e regolare di Ω, denotata con $\mathcal{T}_h$. È possibile dimostrare che [3]:

$$||u - u_h||_{H^1(\Omega)} \leq c_p \sqrt{n} \Big(\sum_{K \in \mathcal{T}_h} |\rho_K(u_h)|^2 \Big)^{\frac{1}{2}}$$

dove c_p è la costante di Pointcaré, n è il massimo di elementi che possono condividere con lo stesso triangolo un vertice o un lato all'interno di $\mathcal{T}_h$ e $\rho_K(u_h)$ è una quantità relativa all'elemento K che può essere calcolata una volta nota u_h.
Vediamo un possibile impiego di tale formula. Se si trascura l'effetto della quantità $c_p\sqrt{n}$ (di difficile approssimazione numerica), imponendo che abbia valora unitario, e si ipotizza che, fissata un'opportuna tolleranza $\varepsilon > 0$, valga:

$$\frac{1}{4}\frac{\varepsilon^2}{N}||u_h||^2_{H^1(\Omega)} \leq |\rho_K(u_h)|^2 \leq \frac{9}{4}\frac{\varepsilon^2}{N}||u_h||^2_{H^1(\Omega)} \qquad \forall K \in \mathcal{T}_h, N = |\mathcal{T}_h| \tag{2}$$

si ottiene il seguente limite superiore per l'errore relativo:

$$\frac{||u - u_h||_{H^1(\Omega)}}{||u_h||_{H^1(\Omega)}} \leq \frac{2}{3}\varepsilon$$

Tale stima è la conseguenza solo della seconda disuguaglianza, che inoltre fornisce condizioni di raffinamento. La prima impone anche un limite dal basso ai residui ed è responsabile del deraffinamento della griglia.

3. Algoritmo di adattività di griglia

L'idea alla base dell'algoritmo è quella di calcolare una succesione di coppie soluzione triangolazione $(u_h, \mathcal{T}_h)$, modificando ad ogni passo la triangolazione, sulla base del valore dei residui della soluzione precedente, al fine di ottenere, ad una certa iterazione, una coppia $(\hat{u}, \hat{\mathcal{T}}_h)$ che rispetti la condizione [2]. Le operazioni di modifica di una triangolazione, messe a disposizione dalla libreria [1] impiegata nel codice preesistente [2], sono le seguenti:

- Rimozione di un vertice.
- Aggiunta di un punto interno ad un elemento.

Utilizzando tali operazioni , è stato formulato il seguente algoritmo:

1 Calcolo di una coppia soluzione triangolazione $(u_h, \mathcal{T}_h)$.
2 Fissata una tolleranza $\varepsilon > 0$, si genera una coppia $(u_h^*, \mathcal{T}_h^*)$ a partire da $(u_h, \mathcal{T}_h)$, tramite le seguenti operazioni:
  2.1 Calcolo di $||u_h||_{H^1(\Omega)}$ e di $\rho_K(u_h)$, $\forall K \in \mathcal{T}_h$.
  2.2 $\mathcal{T}_h^*$ è $\mathcal{T}_h$ con le seguenti modifiche:
    2.2.1 $\forall K$, se $\rho_K(u_h)^2 > \frac{9}{4}\frac{\varepsilon^2}{N}||u_h||_{H^1(\Omega)}$, si aggiunge a $\mathcal{T}_h$ il baricentro.
    2.2.2 Per ogni vertice di $\mathcal{T}_h$, se la media di $\rho_K(u_h)$ sui $K \in \mathcal{T}_h$ incidenti in esso è maggiore di $\frac{1}{4}\frac{\varepsilon^2}{N}||u_h||_{H^1(\Omega)}$, viene eliminato il vertice stesso.
  2.3 Calcolo di u_h^* soluzione associata a $\mathcal{T}_h^*$.
3 Se $\mathcal{T}_h$ non è stata modificata $(\hat{u}, \hat{\mathcal{T}}_h) = (u_h, \mathcal{T}_h)$. In caso contrario $(u_h, \mathcal{T}_h) = (u_h^*, \mathcal{T}_h^*)$ e si ritorna al passo 2.
4 $(\hat{u}, \hat{\mathcal{T}}_h)$ rispetta la condizione (2), quindi è la soluzione cercata.

Bisogna osservare che le scelte effettuate per i punti 2.2.1 (raffinamento) e 2.2.2 (deraffinamento) non sono le uniche possibili. Un'alternativa riguardante il raffinamento è l'aggiunta del punto medio di ogni lato dell'elemento per poi eseguire la suddivisione in quattro elementi. Per il deraffinamento è invece possibile esguire il test con una condizione meno restrittiva, cioè considerare il residuo massimo sugli elementi incidenti invece del valor medio.

4. Implementazione

Prima di procedere, ricordiamo alcune caratteristiche e funzionalità del codice a disposizione che sono state impiegate e, al contempo, hanno influenzato la scelta implementativa per l'algoritmo di raffinamento. Con riferimento al problema modello sopra esposto, la classe `Poisson` [2] richiede, come parametri di costruzione:

- La funzione f come un oggetto del tipo `func`:

  ```
  typedef double (*func)(double const &, double const &);
  ```

- Una lista di punti, il cui inviluppo convesso coincide con Ω, come un oggetto del tipo `PointList`:

  ```
  typedef std::list<Point> PointList;
  ```

  in cui il tipo `Point` rappresenta i punti all'interno della libreria CGAL;
- Un numero reale che rappresenta il diametro massimo che potranno avere gli elementi della triangolazione;

Di seguito la firma della classe `Poisson`:

```
class Poisson(PointList const &boundary, func f, double const & criteria)
```

Il costruttore della classe, utilizzando opportuni metodi, esegue le seguenti operazioni:

- Crea una triangolazione di Delaunay $\mathcal{T}_h$ nel rispetto del parametro h;
- Associa ad ogni nodo di $\mathcal{T}_h$ un indice;
- Assembla la matrice di stiffness e il termine noto;
- Risolve il sistema lineare;
- Salva la soluzione all'interno della triangolazione.

Esporremo le idee base dell'implementazione adottando un approccio del tipo *top down*. Vediamo come sono stati eseguiti i vari passi dell'algoritmo nel codice principale, entrando nel dettaglio dei metodi e delle classi aggiunte, nei prossimi capitoli. Il passo 1 viene eseguito inserendo i dati del problema necessari per la costruzione di un oggetto della classe `Poisson`, creato subito dopo. Come osservato precedentemente, nel momento della costruzione, viene generata la prima coppia soluzione triangolazione. In particolare ricordiamo che la soluzione è contenuta all'interno della struttura dati della triangolazione. Il passo 2 viene eseguito tramite un opportuno metodo della classe `Poisson` chiamato `refine` e descritto nel prossimo paragrafo. Tale metodo riceve in ingresso la tolleranza ε, aggiorna la coppia $(u_h, \mathcal{T}_h)$

in accordo con l'algoritmo e restituisce in uscita il numero di punti aggiunti e tolti alla triangolazione. Per eseguire il passo 3 si usa l'output del metodo `refine` per stabilire se la triangolazione è stata modificata oppure no. Di seguito si riporta la parte di codice principale che esegue l'algoritmo:

```
func f ...;
PointList bound ...;
double h = ...;
double eps = ...;
Poisson p(bound, f, h);
unsigned int inseriti_e_tolti;
unsigned int iterata = 1;
        do {
                inseriti_e_tolti = p.refine(eps);
                ++iterata;
        } while(inseriti_e_tolti);
```

4.1. **Metodo refine.** Come visto in precedenza la possibiltà di aggiungere informazioni a punti e facce della triangolazione era stata sfruttata per la numerazione dei nodi ed il salvataggio della soluzione. In questo caso l'idea è di aggiungere informazioni sulle facce utili per memorizzare il valore dei residui locali ed altre quantità impiegate in seguito. Per completezza riportiamo il codice relativo alle informazioni riguardanti le facce:

```
class FaceInfo {
                /* gradiente della soluzione uh (sui P1) */
                double gradx_;
                double grady_;
                /* ampiezza della faccia */
                double h_;
                /* residuo (stimatore H1) */
                double res_;
};
```

e i nodi:

```
class PointInfo {
```

```
        /* Tipo condizione al contorno */
        enum BConditionType { NONE, DIRICHLET, NEUMANN };
        BConditionType bcond_;
        /* Indice del vertice */
        unsigned int index_;
        /* Valore della soluzione */
        double value_;
        /* Indicatore vertici della frontiera */
        bool bound_list_;
        /* Gradiente della proiezione di Clement */
        double gradCx_;
        double gradCy_;
};
```

Cominciamo con la firma del metodo `refine`:

```
unsigned int Poisson::refine(double const & eps)
```

Di seguito riportiamo parte dell'implementazione dei vari punti del passo 2.

2.1 Il calcolo e salvataggio nella triangolazione dei residui, nonchè il calcolo della norma H^1 della soluzione approssimata, viene effettuato dal metodo `CalcRes`, definito nel prossimo paragrafo.

2.2.1 Per la modifica della triangolazione si creano 2 liste di punti, quelli da aggiungere e quelli da togliere:

```
PointList da_inserire;
std::list<CDT::Vertex_handle> da_eliminare;
CDT::Finite_faces_iterator itF;
CDT::Finite_vertices_iterator itV;
/* Inserimento */
itF = cdt_.finite_faces_begin();
while (itF != cdt_.finite_faces_end()) {
Integrator2::Geometry g(*itF);
if ((itF->info().res() > max_res) && (std::abs(g.det()) > 9.e-4)) {
        double x0 = itF->vertex(0)->point().x();
        double y0 = itF->vertex(0)->point().y();
```

```
            double x1 = itF->vertex(1)->point().x();
            double y1 = itF->vertex(1)->point().y();
            double x2 = itF->vertex(2)->point().x();
            double y2 = itF->vertex(2)->point().y();
    da_inserire.push_back(Point(
            (x0+x1+x2)/3.,
            (y0+y1+y2)/3.
                    ));
            }
            ++itF;
    }
    /* Rimozione */
            itV =cdt_.finite_vertices_begin();
            while (itV != cdt_.finite_vertices_end()) {
    /* Non eliminabili in quanto nodi di bordo*/
                    if (itV->info().isBoundary()) {
                    ++itV;
                    continue;
                    }
    /* Residuo medio */
            double t = 0.;
            unsigned int n = 0;
            CDT::Face_circulator cF = cdt_.incident_faces(itV);
            CDT::Face_circulator end = cF;
            do {
            if (!cdt_.is_infinite(cF)) {
                    t += cF->info().res();
                    ++n;
            }
            ++cF;
            } while(cF != end);
                    t /= n;
    /* Rimozione */
```

```
            if (t < min_res)
                    da_eliminare.push_back(itV);
                    ++itV;
}
```

Usando le due liste si aggiungo e si rimuovono i punti, costruendo così la nuova triangolazione:

```
if (da_eliminare.size()) {
std::list<CDT::Vertex_handle>::iterator it = da_eliminare.begin();
while(it != da_eliminare.end()) {
        cdt_.remove(*it);
        ++it;
}
}
if (da_inserire.size())
cdt_.insert(da_inserire.begin(), da_inserire.end());
/* Costruzione della nuova griglia */
setBConditions();
enumNodes();
```

2.3 A questo punto si ricalcola la soluzione sulla nuova triangolazione:

```
/* Costruzione matrice e termine noto */
Matrix A;
Vector F;
makeMatrixTermineNoto(A, F);
/* Soluzione del sistema lineare */
Vector x(n_point, 0.);
solveSystem(A, x, F);
/* Salvataggio della soluzione nella griglia */
saveSolution(x);
```

Infine il metodo ritorna la somma del numero di nodi inseriti e tolti:

```
return da_inserire.size() + da_eliminare.size();
```

4.2. **Metodo `CalcRes`.** Come accennato precedentemente, tale metodo calcola, oltre che $||u_h||_{H^1(\Omega)}$, i residui $\rho_K(u_h) \forall K$, per poi salvarli come informazioni relative alle facce. Visto che la maggior parte del codice è dedicata al calcolo dei residui ne ricordiamo di seguito l'espressione:

$$\rho_K(u_h) = h_K||f + \Delta_c u_h||^2_L(K) + \frac{1}{2}h_K^{\frac{1}{2}}||[\frac{\partial u_h}{\partial n}]||^2_L(K), \forall K \in \mathcal{T}_h$$

in cui:

- h_K è il diametro di K.
- $\Delta_c u_h = \frac{\partial}{\partial x}u_{x,c} + \frac{\partial}{\partial y}u_{y,c}$, dove $u_{x,c}, u_{y,c} : \Omega \to \mathbb{R}$ sono ricostruzioni lineari delle derivate parziali di u_h (costanti a tratti, essendo u_h lineare). Più in dettaglio, sia $v : \Omega \to \mathbb{R}$ una funzione costante su ogni elemento, si definisce la funzione interpolante di Clèment:

  $$R_h v(x) = \sum_{N_j}(P_j)(N_j)\varphi_j(x)$$

  dove:
  - φ_j è la funzione di base lagrangiana associata al vertice N_j.
  - $P_j v$ è il piano definito sulla patch K_{N_j} (ossia l'unione degli elementi che condividono il vertice N_j) individuato dalla relazione seguente:

    $$\int_{K_{N_j}}(P_j v - v)\Psi \mathrm{d}x = 0,\ \Psi = 1, x, y$$

    Poniamo dunque $u_{x,c} = R_h \frac{\partial}{\partial x}u_h$ e $u_{y,c} = R_h \frac{\partial}{\partial y}u_h$
- $[\frac{\partial u_h}{\partial u}] : \partial K \to \mathbb{R}$, $K \in \mathcal{T}_h$ è definito nel modo seguente:
  - Sia e un lato di K, se $e \in \partial\Omega$ allora $[\frac{\partial u_h}{\partial n}](x) = 0$, $\forall x \in e$.
  - Se $e \notin \partial\Omega$ allora $[\frac{\partial u_h}{\partial n}](x) = (\nabla u_h|_K - \nabla u_h|_{\bar{K}})n$, $\forall x \in e$.

Riferimenti bibliografici

[1] CGAL Editorial Board. *CGAL User and Reference Manual*, 3.4 edition, 2008.

[2] D. Ferrarese e M. Penati. Grid adaptation strategies, 2009.

[3] Alfio Quarteroni. *Modellistica Numerica per Problemi Differenziali*. Springer-Italia, Milan, IT, 2008.