Elementi Finiti - Esercitazione 1
Primi passi in Julia - Differenze Finite

Prof. Giancarlo Sangalli Ivan Bioli

06 Marzo 2025

1 Ripasso sul corso JuliaAcademy

1.1 Pass-by-sharing in Julia

Documentazione ufficiale di Julia su *"Argument Passing Behavior"*:

`docs.julialang.org/en/v1/manual/functions/#man-argument-passing`

In Julia, il passaggio degli argomenti alle funzioni segue la convenzione chiamata **pass-by-sharing**. Questo significa che i valori non vengono copiati quando vengono passati a una funzione, ma gli argomenti della funzione diventano nuovi legami, o *bindings* (cioè nuovi "nomi"), che puntano agli stessi valori passati, proprio come in un'assegnazione `argument_name = argument_value`. Il comportamento del passaggio delle variabili dipende dalla loro mutabilità:

- se il valore è *immutabile* (es. numeri, stringhe, tuple), qualsiasi modifica all'interno della funzione creerà un nuovo oggetto senza alterare l'originale;
- se il valore è *mutabile* (es. array, dizionari), le modifiche effettuate all'interno della funzione saranno visibili anche al di fuori, poiché la funzione opera direttamente sull'oggetto originale, solo tramite un nome diverso.

Esercizio 1

Quale è il risultato stampato a video dalle ultime due righe, nel codice che segue? Verificate la vostra risposta in Julia.

```julia
function f(x, y)
        x[1] = 42
        y = 7 + y
        return y
end

a = [4, 5, 6]
b = 3
f(a, b)

println(a)
println(b)
```

Esercizio 2

Sapreste comparare questo comportamento con quello di altri due linguaggi interpretati come Python e MATLAB? Quali sono i vantaggi dell'approccio di Julia?

Esercizio 3

Quale sarebbe un nome più appropriato per la funzione definita sopra, visto che modifica uno dei propri argomenti?

1.2 Optional Arguments e Keyword Arguments in Julia

Documentazione ufficiale di Julia sull'argomento

- Funzioni: `docs.julialang.org/en/v1/manual/functions/`
- Optional Arguments: `docs.julialang.org/en/v1/manual/functions/#Optional-Arguments`
- Keyword Arguments: `docs.julialang.org/en/v1/manual/functions/#Keyword-Arguments`

1.2.1 Optional Arguments

In MATLAB, i parametri opzionali si gestiscono spesso con `nargin` o `varargin`, mentre i parametri con nome (*name-value pairs*) richiedono una struct o `inputParser`. In Julia, invece, esistono due meccanismi nativi per gestire gli argomenti opzionali:

1. **Optional Arguments**: argomenti con valore predefinito che possono essere omessi.
2. **Keyword Arguments**: argomenti con nome che possono essere specificati in qualsiasi ordine.

Gli **optional arguments** sono parametri con un valore di default, simili a quelli definiti in MATLAB con `nargin`. Ad esempio, in Julia scriviamo:

```julia
function greet(name="utente")
    println("Ciao, ", name, "!")
end

greet()        # Output: Ciao, utente!
greet("Anna") # Output: Ciao, Anna!
```

mentre in MATLAB scriveremmo

```matlab
function greet(name)
    if nargin < 1
        name = "utente";
    end
    fprintf("Ciao, %s!\n", name);
end
```

I **keyword arguments** permettono di passare parametri con nome, simili ai *name-value pairs* in MATLAB. Un esempio:

```julia
function describe_person(name; age=30, city="Roma")
    println("$name ha $age anni e vive a $city.")
end

describe_person("Marco")                  # Marco ha 30 anni e vive a Roma.
describe_person("Elena", age=25)          # Elena ha 25 anni e vive a Roma.
describe_person("Luca", city="Milano")    # Luca ha 30 anni e vive a Milano.
```

Esempio equivalente in MATLAB

```matlab
function describe_person(name, varargin)
    p = inputParser;
    addParameter(p, 'age', 30);
    addParameter(p, 'city', 'Roma');
    parse(p, varargin{:});

    age = p.Results.age;
    city = p.Results.city;

    fprintf("%s ha %d anni e vive a %s.\n", name, age, city);
end
```

Notate come la sintassi di Julia sia molto più facile da imparare e, soprattutto, da comprendere una volta scritta. La Tabella 1 contiene un riassunto di quanto detto su optional arguments e keyword arguments.

Caratteristica	Optional Arguments	Keyword Arguments
Posizione	Deve seguire l'ordine della funzione	Può essere specificato in qualsiasi ordine
Valore predefinito	Sì	Sì
Sintassi	`arg=default`	`; keyword=default`
Flessibilità	Minore	Maggiore

Tabella 1: Tabella riepilogativa su optional arguments e keyword arguments.

Esercizio 4

Scrivere una funzione Julia chiamata `example` che deve:

- Accettare un argomento obbligatorio `x`.
- Avere un parametro opzionale `y` con valore predefinito 10.
- Avere un parametro con nome (*keyword argument*) `z` con valore predefinito 20.
- Restituire la somma di `x`, `y` e `z`.

2 Riscaldamento con Differenze Finite

Questo primo esercizio ha lo scopo di farvi prendere confidenza con il linguaggio di programmazione Julia, iniziando con operazioni di base come: definire una funzione, creare un vettore, assemblare una matrice ed eseguire dei grafici. Rimaniamo nell'ambito delle equazioni alle derivate parziali (PDEs), implementando una soluzione numerica per l'equazione di Poisson monodimensionale utilizzando il metodo delle differenze finite, una tecnica alternativa agli elementi finiti. L'obiettivo è risolvere il problema di Poisson monodimensionale

$$-\Delta u = f, \quad x \in \Omega = (0,1), \tag{1}$$

con condizioni al bordo di Dirichlet (non necessariamente omogenee)

$$u(0) = g_a, \quad u(1) = g_b. \tag{2}$$

Per risolvere numericamente il problema, discretizziamo il dominio $\Omega = (0,1)$ suddividendolo in N intervalli di lunghezza $h = \frac{1}{N}$, i cui punti di griglia sono

$$x_i = ih, \qquad i = 0, 1, \ldots, N.$$

Cerchiamo una approssimazione discreta della soluzione

$$u_i \approx u(x_i) \quad \text{per} \quad i = 0, 1, \ldots, N.$$

Per approssimare l'operatore $-\Delta u(x) = -u''(x)$, utilizziamo lo schema a differenze finite centrate, che prende la forma:

$$-u''(x) \approx \frac{-u(x+h) + 2u(x) - u(x-h)}{h^2}. \tag{3}$$

Imponendo che l'approssimazione (3) sia uguale al termine noto $f(x)$ in tutti i punti interni della griglia e utilizzando le condizioni al bordo (2), otteniamo il seguente sistema di equazioni:

$$\begin{aligned} &u_0 = g_a, \\ &\frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} = f_i, \qquad i = 1, \ldots, N-1, \\ &u_N = g_b. \end{aligned}$$

Queste equazioni possono essere riscritte, per i punti interni, come un sistema lineare della forma:

$$\mathbf{A}\mathbf{u}_h = \mathbf{b},$$

dove la matrice $\mathbf{A}$ e è tridiagonale della forma

$$\mathbf{A} = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \in \mathbb{R}^{(N-1)\times(N-1)},$$

mentre il vettore $\mathbf{b}$ è dato da:

$$\mathbf{b} = \begin{bmatrix} f(x_1) + \frac{g_a}{h^2} \\ f_2 \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) + \frac{g_b}{h^2} \end{bmatrix} \mathbb{R}^{N-1}.$$

Esercizio 5

Implementare in Julia il metodo delle differenze finite descritto sopra. In particolare:

1. Attivare l'environment creato nella lezione precedente. A questo fine, inserire come prime due righe del proprio codice:

```
import Pkg
Pkg.activate("elementifinitiunipv_pkg")
```

2. Caricare i pacchetti necessari: `LinearAlgebra` (per operazioni con vettori e matrici), `SparseArrays` (per gestire matrici tridiagonali in modo efficiente), `Plots` (per la rappresentazione grafica degli errori), `LaTeXStrings` (per scrivere formule in LaTeXnei grafici).

3. Definire due funzioni in Julia: `f` e `u` che calcolino rispettivamente il termine noto $f(x)$ e la soluzione esatta $u(x)$ del problema di Poisson monodimensionale. Si prenda come soluzione esatta $u(x) = \cos(\pi x)$, con corrispondente termine noto $f(x) = \pi^2 \cos(\pi x)$.

4. Implementare una funzione `poisson1d` che prenda in input il numero di intervalli `N`, il termine noto `f`, i dati di Dirichlet `ga` e `gb` e restituisca la soluzione approssimata $\mathbf{u}_h$ del problema di Poisson monodimensionale. Si presti particolare attenzione a utilizzare una rappresentazione sparsa per la matrice $\mathbf{A}$ per garantire efficienza computazionale. A tale fine, si può utilizzare la funzione `spdiagm` del pacchetto `SparseArrays`.

Esercizio 6

È noto che, per il problema in questione, con una discretizzazione a differenze finite l'errore diminuisce quadraticamente rispetto a h, ovvero:

$$\|u_h - u\|_\infty \leq C \cdot h^2.$$

1. Implementare una funzione `compute_error` che prenda in input la soluzione esatta u, la soluzione approssimata u_h e restituisca l'errore $\max_i |u_i - u(x_i)|$.

2. Risolvere il problema di Poisson monodimensionale per $N = 10, 20, 40, 80, 160, 320$ e calcolare l'errore $\max_i |u_i - u(x_i)|$ per ciascuna soluzione approssimata.

3. Realizzare un grafico dell'errore in funzione di h su scala log-log, confrontandolo con una curva di riferimento $\mathcal{O}(h^2)$.

4. Divertirsi a personalizzare il grafico, aggiungendo titoli, legende, griglie, ecc.

5. Salvare il grafico prodotto in formato PDF o PNG.

3 Meshing per Elementi Finiti

Come avrete intuito dall'esercizio precedente, le differenze finite si basano su griglie. Gli elementi finiti, invece, si basano su mesh; in questo corso, nello specifico, su *triangolazioni*, cioè mesh costituite da elementi triangolari. Ma come si descrive una triangolazione? La struttura dati essenziale per rappresentarla è formata dalle coordinate dei vertici e dalla matrice di incidenza.

Per la descrizione di una triangolazione, utilizziamo due matrici:

- `p`: Una matrice di dimensioni $2 \times N_{\text{points}}$, dove ogni colonna rappresenta le coordinate di un vertice. In particolare, la prima riga contiene la coordinata x di ciascun punto, mentre la riga colonna contiene la coordinata y. Qui, N_{points} è il numero di nodi della mesh.

- `T`: Una matrice di dimensioni $3 \times N_{\text{tri}}$, dove ogni colonna rappresenta un triangolo e ogni riga indica l'indice di uno dei tre vertici che compongono il triangolo. Qui, N_{tri} è il numero totale di triangoli nella mesh. Gli elementi della matrice sono interi che indicano gli indici dei vertici corrispondenti nella matrice delle coordinate.

La matrice `T` descrive la connettività della mesh, ossia come i punti sono connessi per formare i triangoli, mentre la matrice `p` memorizza le coordinate spaziali di ciascun vertice.

Vi forniremo gran parte delle routines necessarie per il meshing durante il corso, utilizzando il generatore di mesh Gmsh e il pacchetto Julia `Gmsh.jl`. Le routines si trovano nel file `modules/Meshing.jl` e verranno aggiornate man mano che il corso progredisce.

Esercizio 7

L'obiettivo di questo esercizio è familiarizzare con la struttura dati di una triangolazione. Ecco le prime linee di codice che dovreste aggiungere al vostro script Julia:

```
using Revise
includet("<PATH-TO-FOLDER>/Meshing.jl")
```

- `using Revise`: Questo comando carica il pacchetto `Revise.jl`, che consente di ricaricare automaticamente i file modificati durante la sessione, senza bisogno di riavviare Julia.
- `includet("<PATH-TO-FOLDER>/Meshing.jl")`: Questa riga include il modulo `Meshing.jl`, che contiene tutte le funzioni necessarie per lavorare con le mesh. Sostituite `<PATH-TO-FOLDER>` con il percorso corretto del file `Meshing.jl`.

Una volta che queste righe sono state aggiunte, potrete iniziare a sperimentare con le mesh, utilizzando anche la documentazione delle funzioni nel file `Meshing.jl`.

1. Usando le funzioni `mesh_square`, `mesh_circle` e `get_nodes_connectivity`, eseguire alcune mesh per il quadrato e il cerchio unitario con diverse dimensioni della mesh h.
2. Definire una funzione `plot_mesh` che prende in input la matrice di incidenza `T`, le coordinate dei vertici `p` e disegna la triangolazione. La funzione deve iterare su ciascun triangolo e disegnare i bordi uno per uno. Non è necessario concentrarsi sull'efficienza in questa fase, l'obiettivo è essere sicuri di aver compreso correttamente la struttura dei dati.
3. Confrontare il risultato ottenuto con la vostra funzione di visualizzazione con quello ottenuto tramite il pacchetto `Meshes.jl` utilizzando il seguente codice:

```
import Meshes
mesh = to_Meshes(T, p)
Meshes.viz(mesh, showsegments = true)
```