ESAME DI METODI DEL CALCOLO SCIENTIFICO: PROGETTO 1BIS

Implementazione minilibreria per risoluzione di sistemi lineari su matrici sparse

Author
Fumagalli Gabriele 845108
Radice Alessio 804883

Contents

1 Struttura della libreria

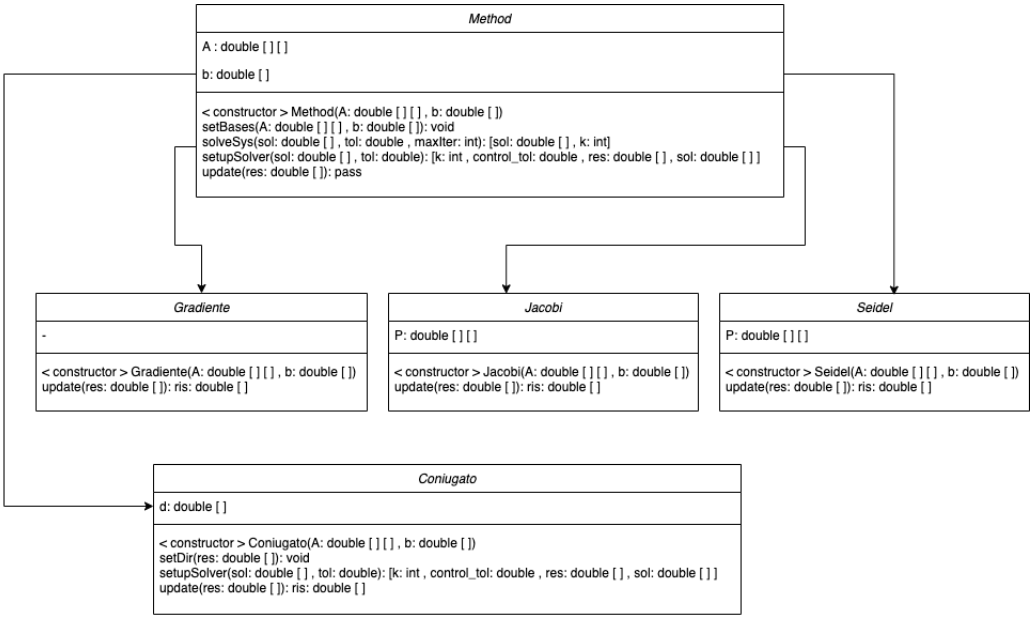

Figure 1: Diagramma UML della libreria

Il progetto, sviluppato su `Python`, fa uso delle librerie `numPy`, per le operazioni vettoriali e matriciali di base (prodotti, norme ecc.) e `sciPy.sparse`, che presenta comandi ottimizzati per matrici sparse.

La libreria implementa una classe base, *Method*, che fornisce il corpo generale di un metodo iterativo: essa si compone di due attributi, ovvero gli elementi del sistema lineare (matrice dei coefficienti A e vettore dei termini noti $\boldsymbol{b}$). Questa scelta è stata pensata per agevolare la replicabilità degli esperimenti: in pratica, nell'oggetto di tipo *Method* è salvato un sistema da risolvere, cosicchè ad ogni chiamata del metodo di risoluzione sia possibile inserire differenti parametri (che nel nostro caso, sono tolleranza, numero massimo di iterazioni e vettore iniziale).

Nello specifico, la classe *Method* fornisce i seguenti metodi:

- `bases(A,b)`: imposta gli attributi della classe.
- il costruttore: richiama il metodo bases, ma solo dopo aver controllato che la matrice A sia simmetrica e definita positiva e $\boldsymbol{b} \neq \boldsymbol{0}$ (altrimenti restituisce un messaggio di errore e il programma si arresta).
- `solveSys(sol,tol,maxIter)`: risolve il sistema lineare $A\boldsymbol{x} = \boldsymbol{b}$ con $\boldsymbol{x}_0 =$ `sol`, con tolleranza fissata a `tol` e numero di iterazioni massimo `maxIter`. Restituisce sia `sol` come il vettore soluzione del sistema, sia il numero `k` di iterazioni eseguite.

```
def solveSys(self, sol, tol, maxIter):
    [k,res,control_tol,sol]=self.setupSolver(sol,tol)
    while (np.linalg.norm(res) >= control_tol)...
    and (k < maxIter):
```

```
        sol += self.update(res)
        res = self.b - self.A*sol
        k += 1
    ris=[sol,k]
    return ris
```

- `setupSolve(sol,tol)`: è la procedura preliminare che setta le condizioni per iniziare il ciclo di iterazioni, ovverosia il numero corrente di iterazione k, il vettore soluzione corrente sol, la tolleranza `controlTol` e il residuo iniziale res. La definizione di default è posta come segue:

```
def setupSolver(self, sol, tol):
    k=0
    res=self.b-self.A*sol
    control_tol=tol*np.linalg.norm(self.b)
    setup=[k,res,control_tol,sol]
    return setup
```

- `update(res)`: restituisce il passo di update in funzione del residuo, è un metodo astratto che rimanda la definizione alle sottoclassi.

Dalla classe *Method*, sono implementate le 4 sottoclassi associate ai metodi iterativi oggetto di analisi: Jacobi (classe *Jacobi*), Gauss-Seidel (classe *Seidel*), gradiente (classe *Gradiente*) e gradiente coniugato (classe *Coniugato*).

1.1 Metodi stazionari

I metodi di Jacobi e Gauss-Seidel prevedono la decomposizione della matrice $A = P + N$ e sfruttano le proprietà di P per effettuare un calcolo rapido dell'aggiornamento

$$update(res) = P^{-1} * res.$$

Nello specifico, la classe *Jacobi* salva in fase di costruzione la matrice $P^{-1} = 1/diag(A)$ della decomposizione di Jacobi e implementa il metodo update col prodotto matrice-vettore.

La classe *Seidel*, in maniera analoga, memorizza la matrice $P = tril(A)$ della decomposizione di Gauss-Seidel e implementa il metodo update restituendo $y = P^{-1}\ res$ come soluzione del sistema triangolare inferiore $Py = res$.

Per effettuare le suddette operazioni, sono state utilizzate le seguenti funzioni del modulo `SciPy.sparse`:

- $A.diagonal()$: salva in un vettore (non sparso) gli elementi sulla diagonale principale della matrice sparsa A
- $diags(v)$: restituisce, in formato sparso, la matrice diagonale con gli elementi del vettore v
- $tril(A)$: estrae la parte triangolare inferiore di A, come matrice sparsa
- $sp\_solveTriangular(P, res, Lower = true)$: del sottomodulo `SciPy.sparse.linalg`, risolve il sistema lineare triangolare $Py = res$ dove P è una matrice in formato sparso.

1.2 Metodi non stazionari

I metodi del gradiente aggiornano la soluzione all'iterazione k nel seguente modo

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \overbrace{\alpha_k \boldsymbol{d}^{(k)}}^{update(res)}$$

dove $res = \boldsymbol{r}^{(k)}$ indica il residuo all'iterazione corrente k.

La classe *Gradiente* mantiene tutte le specifiche di *Method* e definisce solamente l'update coi seguenti valori:

$$\alpha_k = \frac{(\boldsymbol{r}^{(k)})^t \ \boldsymbol{r}^{(k)}}{(\boldsymbol{r}^{(k)})^t \ A \ \boldsymbol{r}^{(k)}} \quad e \quad \boldsymbol{d}^{(k)} = \boldsymbol{r}^{(k)}.$$

La classe *Coniugato* invece aggiunge in fase di costruzione l'attributo d corrispondente alla direzione di discesa, poichè viene aggiornato ad ogni iterazione; ad ogni iterazione sono aggiornati sia $\boldsymbol{x}$ che $\boldsymbol{d}$:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha_k \boldsymbol{d}^{(k)} \quad e \quad \boldsymbol{d}^{(k+1)} = \boldsymbol{r}^{(k+1)} + \beta_k \boldsymbol{d}^{(k)}$$

dove

$$\alpha_k = \frac{(\boldsymbol{d}^{(k)})^t \ \boldsymbol{r}^{(k)}}{(\boldsymbol{d}^{(k)})^t \ A \ \boldsymbol{d}^{(k)}} \quad e \quad \beta_k = \frac{(\boldsymbol{d}^{(k)})^t \ A \ \boldsymbol{r}^{(k+1)}}{(\boldsymbol{d}^{(k)})^t \ A \ \boldsymbol{d}^{(k)}}.$$

La procedura iterativa per il metodo del gradiente coniugato è stata pensata in questo modo:

- $k = 1$: Il metodo del gradiente coniugato inizializza $\boldsymbol{d}^{(0)} = \boldsymbol{r}^{(0)}$, perciò la prima iterazione è quella di un normale metodo del gradiente. Al termine del ciclo avrò aggiornato residuo e soluzione ma non la direzione.
- $k > 1$: Devo aggiornare, nell'ordine, direzione, soluzione e residuo con le formule del metodo del gradiente coniugato. Nel metodo `solveSys` la soluzione è già calcolata prima del residuo, quindi mi basta aggiornare la direzione durante la chiamata del metodo update.

In merito ai metodi di classe, abbiamo dunque:

- override di `setupSolver`: viene sovrascritto con l'esecuzione di un'iterazione fatta col metodo del gradiente e restituisce i valori k (che sarà 0 o 1), il vettore sol=xk, la tolleranza controlTol e il residuo corrente res.

```
def setupSolver(self,sol,tol):
    self.d=self.b-self.A*sol
    iter1=Gradiente(self.A,self.b)
    [sol,k]=iter1.solveSys(sol,tol,1)
    res=self.b-self.A*sol
    control_tol=tol*np.linalg.norm(self.b)
    setup=[k,res,control_tol,sol]
    return setup
```

- `setdir(res)`: aggiorna la direzione di discesa $\boldsymbol{d}^{(k)}$ in funzione del residuo.
- `update(res)`: richiama il metodo `setdir(res)` per aggiornare la direzione e calcola il passo di update.

2 Fase di test

Le matrici in formato .MTX sono lette da `Python` mediante l'istruzione `mmread()` del package `SciPy.IO` e salvate nel formato sparso csr (compressed sparse row). Prima di eseguire i vari metodi, abbiamo raccolto le seguenti informazioni sulle matrici di test.

Matrix A	size	density	condition number	possible structure
spa1	1000	0.182434	2048.1538 ($\simeq 2 \cdot 10^3$)	random
spa2	3000	0.1814776	1411.9678 ($\simeq 1.4 \cdot 10^3$)	random
vem1	1681	0.00473678	324.6439 ($\simeq 3 \cdot 10^2$)	band
vem2	2601	0.00313738	507.0222 ($\simeq 5 \cdot 10^2$)	band

Table 1: Analisi sulle matrici di test

Le matrici non risultano nè eccessivamente malcondizionate nè relativamente dense, in relazione alla loro dimensione; anzi, nel caso delle matrici *vem1* e *vem2* abbiamo un'elevata sparsità e un basso numero di condizionamento. Inoltre, abbiamo cercato di individuare graficamente, per quanto possibile, un'ipotetica struttura delle matrici (comando `matplotlib.pyplot.spy`): le matrici *spa1* e *spa2* non sembrano avere una struttura particolare mentre le matrici *vem1* e *vem2* sembrano essere matrici a bande.

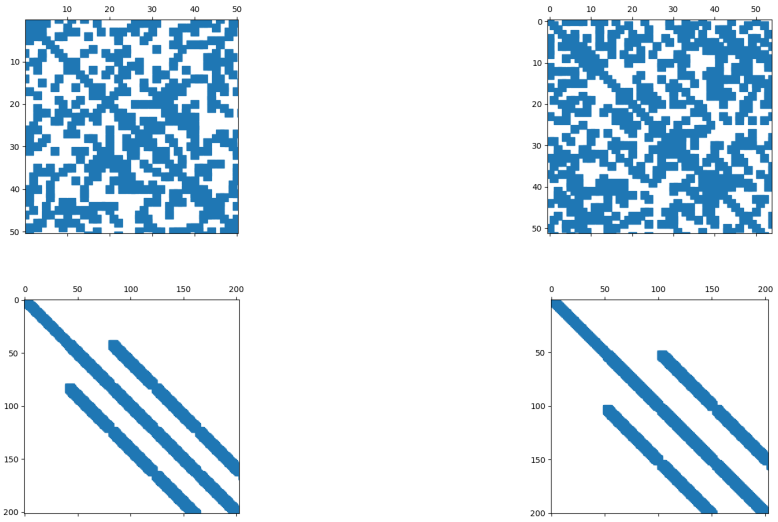

Figure 2: Rappresentazione grafica di una porzione delle matrici mediante comando `spy`. Dall'alto a sinistra, nell'ordine: *spa1*, *spa2*, *vem1*, *vem2*.

Infine, come richiesto in consegna, sono stati risolti i sistemi lineari coi seguenti parametri:

- maxIter=30000, scelto a piacere
- iterazione iniziale $\boldsymbol{x}^{(0)} = \mathbf{0}$
- tolleranze $tol = [10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}]$
- vettori dei termini noti $\boldsymbol{b} = A\boldsymbol{x}$ con $\boldsymbol{x} = [1, 1, ..., 1]$

I risultati sono raccolti nelle tabelle sottostanti.

Table 2: Risultati per la matrice spa1

Errore relativo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	$1.77 \cdot 10^{-3}$	$1.82 \cdot 10^{-2}$	$3.46 \cdot 10^{-2}$	$2.08 \cdot 10^{-2}$
10^{-6}	$1.80 \cdot 10^{-5}$	$1.30 \cdot 10^{-4}$	$9.68 \cdot 10^{-4}$	$2.55 \cdot 10^{-5}$
10^{-8}	$1.82 \cdot 10^{-7}$	$1.71 \cdot 10^{-6}$	$9.82 \cdot 10^{-6}$	$1.32 \cdot 10^{-7}$
10^{-10}	$1.85 \cdot 10^{-9}$	$2.25 \cdot 10^{-8}$	$9.82 \cdot 10^{-8}$	$1.20 \cdot 10^{-9}$

Numero di iterazioni				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	115	9	143	49
10^{-6}	181	17	3577	134
10^{-8}	247	24	8233	177
10^{-10}	313	31	12919	200

Tempo di calcolo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	0.026	0.108	0.061	0.062
10^{-6}	0.040	0.194	1.456	0.126
10^{-8}	0.058	0.274	3.427	0.156
10^{-10}	0.073	0.351	4.932	0.171

Table 3: Risultati per la matrice spa2

Errore relativo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	$1.77 \cdot 10^{-3}$	$2.60 \cdot 10^{-3}$	$1.81 \cdot 10^{-2}$	$9.82 \cdot 10^{-3}$
10^{-6}	$1.67 \cdot 10^{-5}$	$5.14 \cdot 10^{-5}$	$6.69 \cdot 10^{-4}$	$1.20 \cdot 10^{-4}$
10^{-8}	$1.57 \cdot 10^{-7}$	$2.79 \cdot 10^{-7}$	$6.87 \cdot 10^{-6}$	$5.59 \cdot 10^{-7}$
10^{-10}	$1.48 \cdot 10^{-9}$	$5.57 \cdot 10^{-9}$	$6.94 \cdot 10^{-8}$	$5.32 \cdot 10^{-9}$

Numero di iterazioni				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	36	5	161	42
10^{-6}	57	8	1949	122
10^{-8}	78	12	5087	196
10^{-10}	99	15	8285	240

Tempo di calcolo				
tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	0.110	0.221	0.768	0.536
10^{-6}	0.132	0.343	8.565	1.267
10^{-8}	0.185	0.506	23.08	1.915
10^{-10}	0.235	0.611	38.44	2.171

Table 4: Risultati per la matrice vem1

Errore relativo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	$3.54 \cdot 10^{-3}$	$3.51 \cdot 10^{-3}$	$2.70 \cdot 10^{-3}$	$4.08 \cdot 10^{-5}$
10^{-6}	$3.54 \cdot 10^{-5}$	$3.53 \cdot 10^{-5}$	$2.71 \cdot 10^{-5}$	$3.73 \cdot 10^{-7}$
10^{-8}	$3.54 \cdot 10^{-7}$	$3.52 \cdot 10^{-7}$	$2.70 \cdot 10^{-7}$	$2.83 \cdot 10^{-9}$
10^{-10}	$3.54 \cdot 10^{-9}$	$3.51 \cdot 10^{-9}$	$2.71 \cdot 10^{-9}$	$2.19 \cdot 10^{-11}$

Numero di iterazioni				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	1314	659	890	38
10^{-6}	2433	1218	1612	45
10^{-8}	3552	1778	2336	53
10^{-10}	4671	2338	3058	59

Tempo di calcolo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	0.050	12.34	0.044	0.046
10^{-6}	0.088	22.19	0.078	0.048
10^{-8}	0.121	32.79	0.111	0.051
10^{-10}	0.157	42.38	0.142	0.049

Table 5: Risultati per la matrice vem2

Errore relativo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	$4.97 \cdot 10^{-3}$	$4.95 \cdot 10^{-3}$	$3.81 \cdot 10^{-3}$	$5.73 \cdot 10^{-5}$
10^{-6}	$4.97 \cdot 10^{-5}$	$4.94 \cdot 10^{-5}$	$3.79 \cdot 10^{-5}$	$4.74 \cdot 10^{-7}$
10^{-8}	$4.97 \cdot 10^{-7}$	$4.96 \cdot 10^{-7}$	$3.81 \cdot 10^{-7}$	$4.30 \cdot 10^{-9}$
10^{-10}	$4.96 \cdot 10^{-9}$	$4.95 \cdot 10^{-9}$	$3.80 \cdot 10^{-9}$	$2.25 \cdot 10^{-11}$

Numero di iterazioni				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	1927	965	1308	47
10^{-6}	3676	1840	2438	56
10^{-8}	5425	2714	3566	66
10^{-10}	7174	3589	4696	74

Tempo di calcolo				
Tol	Jacobi	Seidel	Gradiente	Coniugato
10^{-4}	0.079	27.01	0.081	0.150
10^{-6}	0.177	50.72	0.146	0.132
10^{-8}	0.233	76.84	0.209	0.121
10^{-10}	0.297	98.90	0.269	0.127

3 Conclusioni

Innanzitutto, tutte le matrici sono simmetriche e definite positive poichè gli oggetti *Method* sono costruiti con successo. Inoltre, in tutti i casi, è evidente come tutti i metodi giungano a convergenza: il numero di iterazioni è sempre largamente inferiore di `maxIter` e gli errori relativi, tenuto conto dei numeri di condizionamento delle matrici, rientrano negli ordini di grandezza attesi.

Notiamo che, il comportamento di un metodo specifico applicato alle matrici *"vem"* (*vem1* e *vem2*) è simile, e vale per tutti i metodi considerati: più precisamente, situazioni critiche su *vem1* si riscontrano anche su *vem2*, solitamente in maniera amplificata dato che *vem2* ha dimensione maggiore. Insieme alle informazioni di Tabella 1 e Figura 2, tutto porta a pensare che vi siano similarità tra le due matrici; un discorso analogo vale per le matrici *"spa"* (*spa1* e *spa2*).

Il metodo del gradiente coniugato è sicuramente il più affidabile, poichè non mostra alcun segno di criticità: per ogni tolleranza e matrice scelta, otteniamo solitamente la soluzione più precisa, in poche iterazioni e tempi più che accettabili. Essendo le matrici simmetriche e definite positive, i teoremi garantiscono la convergenza in un numero di iterazioni non superiore alla dimensione della matrice del sistema. Le performance migliori (particolare enfasi sull'errore relativo) sulle matrici *"vem"* possono essere giustificate dall'elevata sparsità delle matrici.

Il metodo del gradiente è il metodo che effettua il maggior numero di iterazioni con le matrici *"spa"* e il tempo di calcolo complessivo, infatti, ne risente in particolare sulla matrice *spa2*. Una ragione plausibile è che la struttura delle matrici *"spa"* determini delle direzioni di discesa non ottimali ad ogni iterazione, il che spiegherebbe i risultati migliori del metodo del gradiente coniugato; inoltre, la relativa densità delle matrici *"spa"*, potrebbe influire sul tempo di calcolo delle singole iterazioni. Con le matrici *"vem"*, invece, il metodo del gradiente è veloce e preciso, seppur necessiti di parecchie iterazioni in confronto al metodo del gradiente coniugato.

Il metodo di Jacobi opera bene sulle matrici *"spa"*: con la matrice *spa2*, in particolare, supera il metodo del gradiente coniugato in termini di performance. Il metodo non presenta particolari criticità anche applicato alle matrici "vem", ma risulta meno efficiente del metodo del gradiente.

Il metodo di Gauss-Seidel è il metodo migliore da usare sulle matrici *"spa"* per il numero esiguo di iterazioni impiegate a raggiungere convergenza. Nonostante ciò, il tempo totale risulta superiore al metodo di Jacobi. Questo è molto evidente nei risultati ottenuti con le matrici *"vem"*: il metodo, nonostante le poche iterazioni in confronto a Jacobi, impiega tempi molto lunghi. Questo suggerisce una lentezza di calcolo della singola iterazione: la ragione più probabile sta in una difficoltosa risoluzione del sistema lineare ad ogni iterazione, passaggio assente negli altri metodi.

In estrema sintesi, i metodi stazionari si prestano meglio ad operare con le matrici di tipo *"spa"*, mentre i metodi non stazionari lavorano meglio su quelle di tipo *"vem"*.