Laboratorio Didattico di Matematica Computazionale

a.a. 2020-2021

Esercitazione 7

Matrici sparse e Game of Life

Per questa esercitazione verrà corretto l'**esercizio 2**. Create un file `.tar` o `.zip` contenente lo script che risolve l'esercizio e le eventuali function ausiliarie, e caricatelo sulla pagina di e-learning del corso. Se non avete function ausiliarie, potete caricare solo lo script.

In alcune applicazioni si lavora con matrici che contengono molti elementi nulli. Matrici di questo tipo sono dette *sparse*.

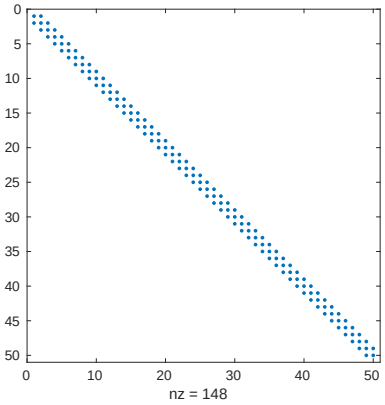
La definizione rigorosa di matrice sparsa può variare a seconda del contesto; generalmente una matrice $n \times n$ si considera sparsa se ha un numero di elementi non nulli dell'ordine di $O(n)$. Per fare un esempio, secondo questa definizione le matrici diagonali sono sparse (perché il numero di elementi potenzialmente non nulli è n), mentre le matrici triangolari superiori non sono sparse (perché il numero di elementi potenzialmente non nulli è $n(n+1)/2$).

1. Matrici sparse in MATLAB

Il comando `spy` permette di visualizzare lo *sparsity pattern* di una matrice, cioè dà una rappresentazione grafica bidimensionale della matrice in cui gli elementi non nulli sono rappresentati da punti colorati, mentre gli elementi nulli sono bianchi. Facciamo un esempio: costruiamo una matrice tridiagonale 50×50 e visualizziamone lo sparsity pattern.

```
diagonale=1:50;
sopradiag=-ones(1,49);
sottodiag=ones(1,49);
A=diag(diagonale)+diag(sopradiag,1)+diag(sottodiag,-1);
spy(A)
```

Dovreste ottenere una figura come la seguente:



in cui si vede chiaramente la struttura tridiagonale (e quindi sparsa) della matrice.

MATLAB dispone di un formato `sparse` apposito per memorizzare matrici sparse in modo più compatto rispetto al formato abituale. Una matrice in formato `sparse` viene salvata come un elenco degli indici degli elementi non nulli e dei rispettivi valori.

La matrice tridiagonale `A` definita sopra è in formato "pieno". Per convertirla in formato `sparse` e vedere come viene rappresentata scrivete nella finestra di comando:

```
A=sparse(A)
```

La conversione da formato sparso a formato pieno si esegue invece con il comando `full`:

```
A=full(A)
```

Il comando `sparse` serve non solo per convertire una matrice da formato pieno a formato sparso, ma anche per creare direttamente una matrice in formato sparso. Per esempio, se scriviamo

```
B=sparse(7,8)
```

otteniamo una matrice 7×8 di zeri, in formato sparso. Possiamo poi modificarne gli elementi con una sintassi simile a quella usata per le matrici piene. Per esempio, per inserire un elemento pari a π in posizione $(2, 3)$ possiamo scrivere

```
B(2,3)=pi
```

Per creare invece una matrice sparsa con alcuni elementi non nulli possiamo usare i comandi seguenti:

```
indici_riga=[1, 3, 5];
indici_colonna=[2, 4, 6];
valori=[pi, exp(1), 42];
C=sparse(indici_riga, indici_colonna, valori, 7, 8)
```

Il comando `nnz` applicato ad una matrice sparsa restituisce il numero di elementi non nulli: se scrivete `nnz(C)` dovreste ottenere `3`.

Il formato sparso è utile non solo per rappresentare matrici sparse, ma anche per eseguire più velocemente operazioni che mantengono la sparsità (cioè il cui output è costituito da matrici sparse, anche se non necessariamente con lo stesso sparsity pattern dell'input).

Consideriamo per esempio la fattorizzazione LU di una matrice quadrata data A. Questa consiste nello scrivere A come prodotto di una matrice L per una matrice U, dove L è triangolare inferiore con elementi diagonali uguali a 1, e U è triangolare superiore. Come probabilmente sapete, la fattorizzazione LU è legata al procedimento di eliminazione di Gauss, e non tutte le matrici ammettono una tale fattorizzazione.

Il calcolo della fattorizzazione LU si esegue in MATLAB con il comando `lu`, che ha la sintassi seguente:

```
[L,U]=lu(A)
```

sia per matrici piene che per matrici sparse. Se A è tridiagonale e fattorizzabile, le matrici L e U risultano bidiagonali. Ci chiediamo quindi se il calcolo della fattorizzazione LU sia più veloce per matrici in formato sparso rispetto a matrici in formato pieno.

Nell'esercizio seguente misureremo il tempo di calcolo della fattorizzazione LU. Questo si può fare con il comando `tic toc`:

```
tic
[L,U]=lu(A);
time_lu=toc;
```

Ora la variabile `time_lu` contiene il tempo (in secondi) impiegato da MATLAB a calcolare la fattorizzazione.

Esercizio 1 *Scrivere uno script Matlab che, per* `n=100:100:1000`*,*

- *crei la matrice tridiagonale* `A`*, di dimensioni* `n`$\times$`n`*, in formato pieno, avente gli elementi diagonali uguali a* 1*, gli elementi sottodiagonali uguali a 0.5 e gli elementi sopradiagonali uguali a 0.1;*
- *esegua la fattorizzazione LU di* `A`*, misuri il tempo impiegato e lo salvi in un apposito vettore* `time_full`*,*
- *converta* `A` *in formato sparso,*
- *ripeta la fattoriazzazione, misuri il tempo impiegato e lo salvi in un apposito vettore* `time_sparse`*,*
- *rappresenti i due vettori dei tempi in un unico grafico.*

Che cosa notate? Quanto valgono i rapporti fra i tempi misurati?

2. The Game of Life

Il Game of Life fu ideato da John Conway (1937-2020) e reso popolare da Martin Gardner negli anni '70.

Il gioco è ambientato in un universo costituito da una griglia bidimensionale infinita composta da celle quadrate. Ogni cella ha due stati possibili: viva oppure morta. Il gioco comincia con una configurazione data di celle vive e morte ed evolve con tempo discreto. Ad ogni passo del gioco, lo stato di ciascuna cella è determinato dall'interazione con le otto celle vicine, secondo le regole seguenti:

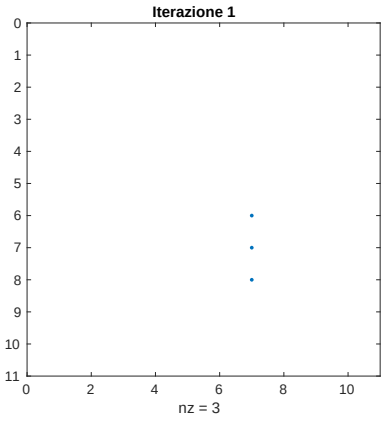
1. se una cella viva ha esattamente due o tre vicine vive, sopravvive;
2. se una cella morta ha tre vicine vive, prende vita;
3. tutte le celle vive che non soddisfano (1) muoiono;
4. tutte le celle morte che non soddisfano (2) restano morte.

Queste regole possono essere interpretate come condizioni sulla densità di popolazione che favoriscono o impediscono la sopravvivenza. Un modo alternativo per riassumerle è il seguente: al prossimo passo del gioco saranno vive tutte e sole le celle che (a) sono vive e hanno esattamente due vicine vive, o (b) hanno esattamente tre vicine vive.

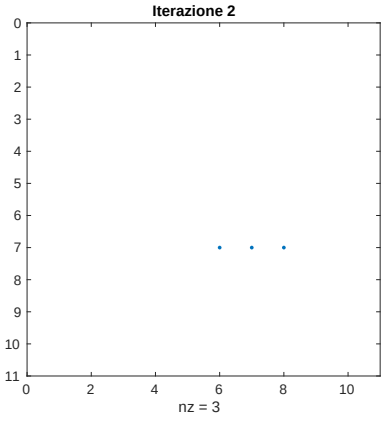
Ci proponiamo di scrivere un'implementazione MATLAB del Game of Life usando le matrici sparse. Come universo prenderemo una matrice sparsa $N \times N$, per un opportuno valore di N, in cui le celle morte sono rappresentate da elementi nulli, mentre le celle vive sono rappresentate da elementi uguali a 1. Il comando `spy` ci fornisce quindi una rappresentazione grafica della configurazione attuale. Facciamo un esempio di configurazione iniziale in un universo 10×10:

```
X=sparse(10,10);
X(6:8,7)=1;
```

Graficamente questa configurazione si rappresenta come nella figura seguente:



Applicando le regole del gioco si vede che al prossimo passo saranno vive solo le celle in posizione $(7, 6)$, $(7, 7)$ e $(7, 8)$:



Al passo seguente otterremo una configurazione uguale a quella iniziale. Quindi, con questi dati iniziali, il gioco procede alternando le due configurazioni. Ovviamente, con dati iniziali diversi possono prodursi fenomeni più interessanti!

Esercizio 2 *Scrivere un'implementazione MATLAB del Game of Life che, dopo aver fissato le dimensioni dell'universo, una configurazione iniziale e il numero di passi* `k` *da eseguire, calcoli e rappresenti graficamente la configurazione ad ogni passo del gioco. (Potete scrivere una function che prenda in ingresso dimensioni, configurazione iniziale e numero di passi, e poi chiamarla da un apposito script, oppure scrivere un unico script che contenga condizioni iniziali e simulazione del gioco). Sperimentate la vostra implementazione con* $N = 100$ *(o più grande) e le due configurazioni iniziali specificate nei file* `glider.txt` *e* `glidergun.txt`.

Per programmare il Game of Life è necessario, ad ogni passo, contare quanti vicini vivi ha ciascuna cella della porzione di universo considerata. Cercate di implementare questa operazione in modo vettoriale!

Suggerimento: dovremo "traslare" la matrice universo `X` *nelle 8 direzioni possibili e sommare tutte le matrici traslate, ottenendo così una matrice* `Y` *che ha come elemento* (i, j) *il numero di vicini vivi della cella di posizione* (i, j) *in* `X`.

Useremo poi degli operatori logici applicati alle matrici `X` e `Y` per determinare la configurazione al prossimo passo.

È consigliabile dare l'istruzione `drawnow` dopo il comando `spy` per aggiornare il grafico ad ogni passo e vedere la successione di configurazioni.

Esercizio 3 *Per rendere più gradevole la rappresentazione grafica dell'Es.2 possiamo creare un'animazione. Sia* `j` *l'indice di iterazione; aggiungete l'istruzione* `F(j)=getframe;` *dopo il comando* `spy` *per creare una variabile* `F` *di tipo* `struct` *contenente le immagini delle varie configurazioni, che potranno poi essere visualizzate come fotogrammi di un'animazione con il comando* `movie(F)`. *Consultate l'help del comando* `movie` *per vedere le opzioni possibili.*

Per saperne di più: trovate un piacevole articolo sul Game of Life nel Giornalino degli Open Days, n.11 (ottobre 2020):
`https://www.dm.unipi.it/webnew/sites/default/files/orientamento/Giornalino_11.pdf`