|  |  |
| --- | --- |
|  | Francesco Cavallini  Matricola: 920835  f.cavallini8@campus.unimib.it  Corso di studi di Informatica Magistrale |
| Università degli studi Milano Bicocca  Milano, Padiglione U24  Ottobre, 2024 |  |

|  |  |
| --- | --- |
|  | |
| Easy Matrix  Metodi del Calcolo Scientifico  Relazione Progetto 1-Bis  **Consegna:**  **“““**  Si utilizzi un linguaggio di programmazione a vostra scelta: c++, fortran, java, pyton, etc. Lo scopo del progetto è quello di implementare una mini libreria che esegua i seguenti solutori iterativi, limitatamente al caso di matrici simmetriche e definite positive:  (1) metodo di Jacobi;  (2) metodo di Gauß-Seidel;  (3) metodo del Gradiente;  (4) metodo del Gradiente coniugato.  **”””**  Riferimenti : | |
| Repository git-hub: | https://github.com/VR3ED/Easy\_Matrix |
| Download libreria: | <https://www.nuget.org/packages/easy_matrix/1.0.0> |

Sommario

[0. Introduzione 3](#_Toc1)

[0.1. Obbiettivi 3](#_Toc2)

[0.2. Scelte implementative 3](#_Toc3)

[1. Parte-1: Sviluppo della libreria 4](#_Toc4)

[1.1. Requisiti: 4](#_Toc5)

[1.2. Le basi per lo sviluppo: 4](#_Toc6)

[1.3. Implementazione JacobiSolver 9](#_Toc7)

[1.4. Implementazione GaussSeidelSolver 11](#_Toc8)

[1.5. Implementazione GradientSolver 13](#_Toc9)

[1.6. Implementazione GradientSolver 15](#_Toc10)

# 0. Introduzione

## 0.1. Obbiettivi

Come definito nella pagina iniziale, l’obbiettivo del progetto è quello di implementare una mini libreria che esegua una serie di solutori iterativi, limitatamente al caso di matrici simmetriche e definite positive.

## 0.2. Scelte implementative

Si sceglie di suddividere lo sviluppo del progetto in 2 parti:

1. Implementazione della libreria
2. Paragoni con librerie open source ed analisi dei risultati.

Per lo sviluppo della prima parte si sceglie di usare il linguaggio C# utilizzando il framework .Net 8.0 per i seguenti motivi:

* **Facilità di pubblicazione libreria:**

Microsoft offre la possibilità di pubblicare gratuitamente la propria libreria sviluppata usando il linguaggio C# sulla piattaforma [NuGet](https://www.nuget.org/). Da questa piattaforma la suddetta libreria potrà venire scaricata, incorporata ed utilizzata in qualsiasi progetto tramite un semplice click al seguente link: <https://www.nuget.org/packages/easy_matrix/1.0.0>

* **Gestione alta precisione:**

Il linguaggio C# permette la gestione dei numeri a virgola mobile con il tipo decimal. Il tipo decimal, seppure permetta di rappresentare solo numeri più piccoli, ha la massima precisione tra i tre tipi a virgola mobile:

|  |  |  |
| --- | --- | --- |
| **C# type/keyword** | **Approximate range** | **Precision** |
| *float* | ±1.5 x 10−45 to ±3.4 x 1038 | ~6-9 digits |
| *double* | ±5.0 × 10−324 to ±1.7 × 10308 | ~15-17 digits |
| *decimal* |  | ~28-29 digits |

Per lo sviluppo della seconda parte si sceglie di fare paragoni con performance di librerie esterne utilizzando il linguaggio python in quanto presenta un amplia gamma di librerie e (come vedremo) codice open-source che renderanno possibile il paragone con la libreria sviluppata al punto 1. Inoltre verrà utilizzato python anche per la formulazione di grafici per il paragone delle statistiche di performance tra libreria sviluppata e codice open source.

# 1. Parte-1: Sviluppo della libreria

## 1.1. Requisiti:

I requisiti che rispetteremo per lo sviluppo delle librerie sono i seguenti:

1. La libreria deve essere in grado di operare con ciascuno dei metodi iterativi sopra menzionati. Inoltre, in caso di utilizzo di librerie esterne, queste librerie devono fornire solamente la struttura dati di matrici e vettori, senza utilizzare i metodi relativi alla risoluzione dei sistemi lineari già implementati al suo interno.
2. I metodi iterativi devono partire da un vettore iniziale nullo e arrestarsi quando la k-esima iterata x(k) soddisfa la condizione:

Dove tol è una tolleranza assegnata dall’utente. Per eseguire i nostri test (che mostreremo nella parte-2) verranno usate . Alternativamente, laddove non si dovesse raggiungere la convergenza entro un numero massimo di iterazioni (maxIter, dove maxIter è un numero elevato a scelta (non inferiore a 20000), la computazione del metodo iterativo dovrà cessare e restituire in output:

convergence = false

1. La libreria deve avere un’architettura ben strutturata anziché essere una sequenza di funzioni indipendenti.

## 1.2. Le basi per lo sviluppo:

Nell’introduzione abbiamo accennato al fatto che tutte le matrici () che daremo in pasto agli algoritmi di libreria sono simmetriche e definite positive; abbiamo quindi che per ogni matrice () per essere **simmetrica e definita positiva** deve rispettare i seguenti criteri:

* La matrice () è simmetrica (ossia )
* La matrice A è definita positiva (ossia per ogni vettore colonna , soddisfa la seguente condizione:

Il metodo di Cholesky permette di scomporre una matrice simmetrica e definita positiva () in

Dove ed sono rispettivamente una matrice triangolare inferiore e superiore

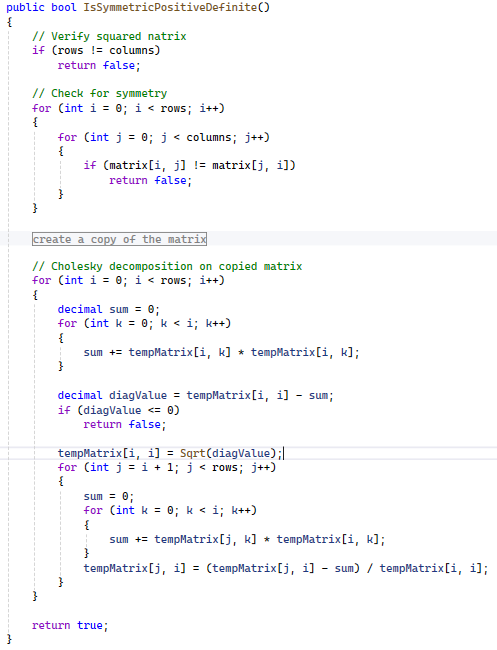
**NOTA**: Metodo di Cholesky

Nella libreria, all’iterno della classe AccurateMatrix (che serve per rappresentare una matrice in formato decimal), è stata infatti inserita una funzione apposta per verificare se le matrici lette in input sono simmetriche e definite positive, ma al posto che controllare se la matrice è definita positiva, si prova invece ad applicare Cholesky, se non ci si riesce allora vuol dire che la matrice non è simmetrica e definita positiva:

Possiamo ora calcolare:

* R[i,i] =

In questo modo si calcola Cholesky per ogni iterazione

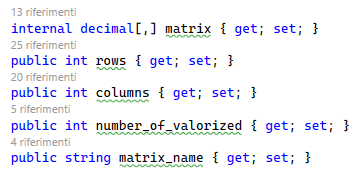


Verifica se l'elemento diagonale calcolato nella decomposizione di Cholesky è positivo, ossia prima si calcola poi si calcola il quadrato del valore diagonale: (se questo valore è >0 possiamo procedere)

Verifica se la matrice è simmetrica comparando

con dentro un doppio loop

all’interno della classe AccurateMatrix, oltre al metodo appena ovviamencritdescritto sono ovviamente presenti i seguenti attributiattriseguen il quale scopo è auto-esplicativo:esplauto-



In C# non esistono metodi per leggere un file .mtx ed assegnare il valore ad un tipo decimal, quindi, in verità, prima di definire se una matrice è simmetrica e definita positiva, la prima sfida implementativa è stata trovare una soluzione per permettere alla libreria di poter leggere i file.

Questa sfida è stata poi superata utilizzando il multi-threading per leggere il file riga per riga come fosse un file txt e creare un parser custom che permettesse di convertire la stringa letta in un valore di tipo decimal (e poi memorizzare il valore nella cella corretta della matrice).   
  
Questo metodo di lettura della matrice non verrà però approfondito in questa relazione in quanto si reputa essere fuori dallo scope della relazione stessa, ma si possono trovare più informazioni su questo metodo sulla repository di git-hub ([qui](https://github.com/VR3ED/Easy_Matrix/blob/main/EasyMatrix/AccurateMatrix.cs))

**NOTA IMPLEMENTATIVA**: Lettura delle matrici da file .mtx

Avendo quindi la possibilità di leggere file .mtx e verificare che le matrici lette siano simmetriche e definite positive il prossimo passo è quello dell’implementazione dei **solutori iterativi**. Più nello specifico un solutore iterativo è un metodo particolare utilizzato per risolvere sistemi di equazioni lineari (rappresentati dalle nostre matrici). Questi metodi generano una successione di approssimazioni (iterativamente) per trovare la soluzione del sistema lineare, migliorando iterazione per iterazione la stima di soluzione; appunto la caratteristica distintiva dei metodi iterativi è che la stessa operazione (o uno stesso insieme di operazioni) viene applicata ripetutamente a ogni iterazione per migliorare l’approssimazione. La convergenza dei metodi iterativi viene raggiunta quando viene raggiunta la soluzione esatta:

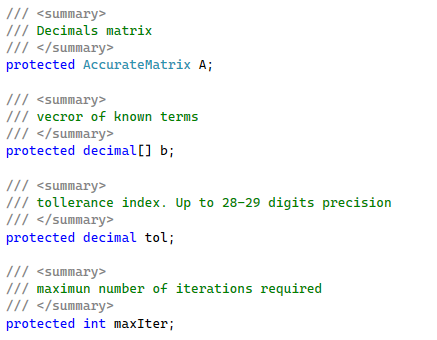
oppure quando (all’iterazione ) viene raggiunta un’approssimazione abbastanza vicina alla soluzione esatta:

Ossia si calcola l’errore relativo e si verifica che questo sia al di sotto di una certa soglia di tolleranza (come anche richiesto da consegna).

Abbiamo quindi che tutti i metodi iterativi che devono venire implementati hanno lo stesso comportamento comune di base:

* Ciclare fino a raggiungere (oppure fino all’iterazione )
* All’interno del ciclo:
  + Calcolare un vettore delle incognite approssimato (la quale logica di calcolo cambia per ogni metodo iterativo) necessaria per calcolare l’errore relativo.
  + Verificare la condizione di uscita (ossia verificare se ) , in caso positivo interrompere il ciclo e mandare in output il vettore delle incognite approssimato (in modo da risolvere il sistema lineare come )

Abbiamo, dunque, che per implementare tutti i solutori iterativi richiesti possiamo sfruttare le proprietà del linguaggio di programmazione C# ed implementare una classe astratta (che non può essere istanziata) chiamata IterativeSolver con i seguenti attributi:



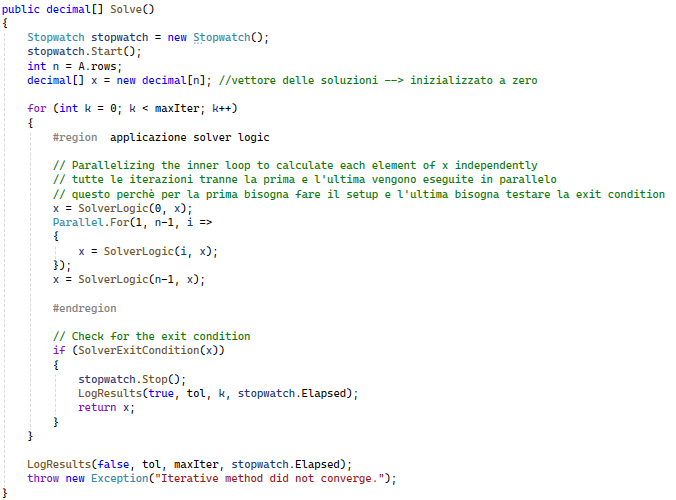
Inoltre la classe implementa al suo interno il metodo Solve() che descrive il comportamento generale di un solutore iterativo (appena descritto):

Nella funzione SolverExitCondition si calcola l’errore relativo e lo si paragona con . Se questa funzione ritorna true allora il metodo iterativo converge e si ritorna in output il vettore

Per i da 0 a (dove n=numero di righe della matrice) si calcola una cella del vettore delle incognite approssimato, ossia si calcola utilizzando la SolverLogic nativa di ogni classe figlia

Se entro MaxIter iterazioni il metodo non converge si ritorna in un eccezione per segnalare la non convergenza

Si cicla da 0 a maxIter, dove, per consegna, maxIter è un parametro determinato dall’utente per avere un numero di iterazioni massime



Si vuole fare notare che il metodo precedentemente descritto non è astratto, il che vuol dire che tutte le classi solver figlie della classe IterativeSolver erediteranno anche questo metodo. Ma, invece, i metodi SolverLogic e SolverExitCondition sono astratti, il che vuol dire che ogni classe figlia di IterativeSolver dovrà implementarli.

Mantenendo questa struttura abbiamo quindi che ogni solutore eseguirà la stessa struttura base di IterativeSolver per eseguire il metodo Solve ma applicherà una logica di calcolo di soluzione approssimata diversa per ogni classe figlia.

Come abbiamo detto precedentemente, anche SolverExitCondition fa’ parte della logica condivisa da tutti i solver iterativi, eppure lo abbiamo definito come metodo astratto, il che vuol dire che ogni classe dovrà ri-implemetare il metodo in questo metodo.

Sembrerebbe una contraddizione, invece è semplicemente stato necessario definire questo metodo in questo modo perché (nonostante tutte le condizioni di uscita rimarranno uguali per tutti i solutori) alcuni solutori hanno bisogno di eseguire delle operazioni di setup per la prossima iterazione all’interno di questo metodo.

Pertanto, nonostante tutte le exit condition di tutti i solutori rimarranno uguali, avremo che si potrebbero presentare delle piccole differenze all’interno dei metodi di SolverExitCondition per puri motivi tecnici di limiti di flessibilità del linguaggio C#

**NOTA IMPLEMENTATIVA**: Perché PerSolverExitCondition è un metodo astratto?

Una volta chiarita qual’è la struttura base di ogni solutore utilizza per trovare una soluzione (con il metodo Solve) possiamo ora passare alla descrizione vera e propria dei metodi SolverLogic e SolverExitCondition delle 4 classi che estendo il comportamento di IterativeSolver, ossia le classi:

* JacobiSolver
* GaussSeidelSolver
* GradientSolver
* ConjugateGradientSolver

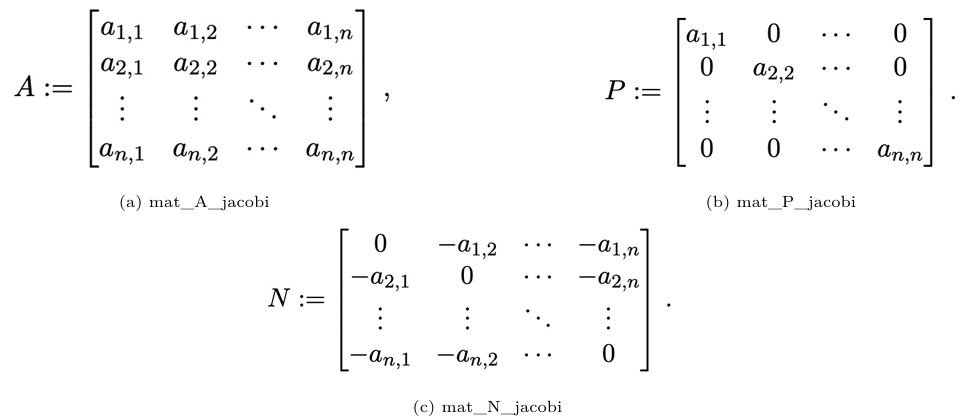
## 1.3. Implementazione JacobiSolver

L’idea di base del metodo di Jacobi è quella di risolvere ogni equazione del sistema per ciascuna delle incognite, esprimendo ogni incognita in termini delle altre incognite e dei termini noti, aggiornando poi ad ogni iterazione i valori calcolati ().

Abbiamo infatti che il metodo di Jacobi sfrutta una scomposizione di (detta **slitting**) che permette di suddividere la matrice in 2 matrici più facili da memorizzare:

dove:

* : Matrice diagonale contenente gli elementi diagonali di .
* : Matrice contenente gli elementi non diagonali di , cioè .



In seguito a questa scomposizione si ha che il sistema può essere scritto come:

Abbiamo infatti che la formula di per ottenere in seguito alla scomposizione è:

dove:

* : vettore delle incognite all'iterazione .
* : vecchio vettore delle incognite all’iterazione precedente ()

Il che vuol dire che se vogliamo computare (per ogni iterazione ) ogni componente allora possiamo aggiornarlo separatamente con questa formula:

Nota che qui il segno è cambiato perché si stanno usando i valori di non di

dove:

* : Termine diagonale della matrice , che corrisponde a .
* : Termine non diagonale. Che corrisponde a

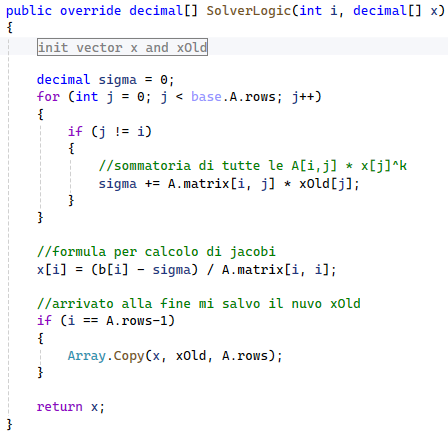
Se andiamo a visualizzare quindi l’implementazione del metodo SolverLogic all’interno della classe JacobiSolver possiamo vedere che implementerà esattamente la formula di aggiornamento di ogni componente appena descritta:

Aggiornamento della variabile ‘xOld’ che corrisponde a per la prossima iterazione

Calcolo aggiornamento :

Calcolo della sommatoria

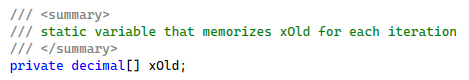
E salvataggio nella variabile ‘sigma’



Calcolo della sommatoria

E salvataggio nella variabile ‘sigma’

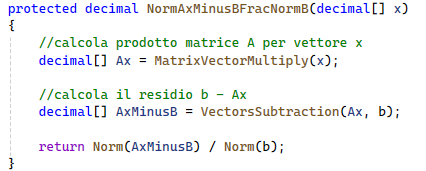
Nota che per permettere ad ogni di avere lo stesso valore di xOld settato alla fine del calcolo si salva questo valore come attributo della classe:



Siccome poi non abbiamo alcuna operazione aggiuntiva da fare per la prossima iterazione il metodo SolverExitCondition sarà semplicemente:



Dove il metodo NormAxMinusBFracNormB (inserito nella classe IterativeSolver come metodo di supporto) è semplicemente la formula come mostrato di seguito:



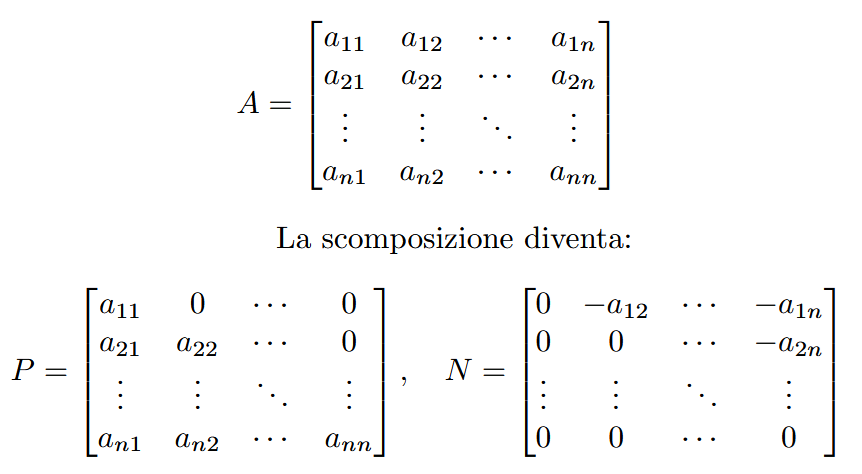
## 1.4. Implementazione GaussSeidelSolver

L’idea di base del metodo di Gauss Seidel è la stessa di Jacobi, con la differenza che (al posto di risolvere ogni equazione del sistema per ciascuna delle incognite) partire dalla prima riga in maniera uguale alla precedente, ma poi vogliamo prendere l'incognita già risolta alla prima riga e la portiamo alla equazione successiva nel sistema, continuando così a catena per tutte le righe dell’equazione.

Quello che succede, di fatto, è che anche qui abbiamo una scomposizione della matrice in 2 matrici più facili da memorizzare:

dove:

* : matrice invertibile che contiene gli elementi diagonali di e la parte strettamente inferiore .
* : matrice con la parte strettamente superiore .



Anche qui, in seguito a questa scomposizione si ha che il sistema può essere scritto come:

Abbiamo infatti che la formula di per ottenere in seguito alla scomposizione è:

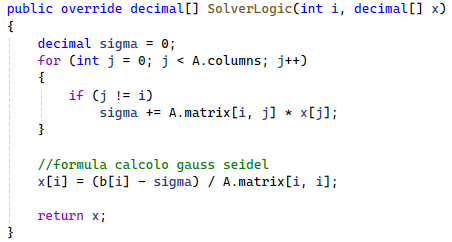
Esattamente come prima, ma avendo però che le matrici ed sono differenti abbiamo che la formula di aggiornamento (per ogni iterazione ) di ogni componente , del vettore delle incognite, cambia, passando a questa formula:

dove:

* La prima sommatoria () usa i nuovi valori aggiornati () per le variabili già calcolate.
* La seconda sommatoria () usa i vecchi valori () per le variabili ancora da calcolare.

Se andiamo a visualizzare quindi l’implementazione del metodo SolverLogic all’interno della classe GaussSeidelSolver possiamo vedere che implementerà esattamente la formula di aggiornamento di ogni componente appena descritta:

Calcolo aggiornamento :



Calcolo delle due sommatorie dinamicamente:

Dove:

* La variabile utilizza i nuovi valori quando , perché questi valori sono già stati aggiornati nel ciclo corrente.
* La variabile utilizza i vecchi valori quando , perché questi valori non sono ancora stati aggiornati in questa iterazione.

Siccome poi non abbiamo alcuna operazione aggiuntiva da fare per la prossima iterazione il metodo SolverExitCondition sarà identico al precedente, ossia:



Dove il metodo NormAxMinusBFracNormB lo stesso in quanto inserito nella classe IterativeSolver dal quale entrambi questi solutori visti fin’ora ereditano.

## 1.5. Implementazione GradientSolver

Il metodo del gradiente si basa sull’interpretazione della risoluzione di un sistema lineare come la ricerca del minimo di una funzione quadratica. La funzione da minimizzare infatti è:

Il minimo di corrisponde alla soluzione del sistema . Mentre la funzione di aggiornamento è:

Dove:

* è il residuo

Più nello specifico, per arrivare risolvere il problema di minimizzazione, abbiamo che per ogni iterazione (prima del controllo della convergenza si verificano queste operazi):

1. **Calcolo del residuo:** Si inizia con una stima iniziale del vettore delle incognite ​. Con questo si calcola il residuo , che rappresenta l'errore assoluto tra il valore attuale e il vettore della soluzione .
2. **Direzione della discesa del gradiente**: Il residuo appena calcolato si usa come direzione di discesa, poiché corrisponde al gradiente della funzione valutato in :

abbiamo che questo gradiente punta nella direzione di crescita massima della funzione , quindi si ha che, per minimizzare la funzione, dobbiamo muoverci in direzione

1. **Aggiornamento :** Si cerca il miglior passo lungo la direzione minimizzando , ottenendo:

calcolato alpha si può poi aggiornare la soluzione:

abbiamo dunque che ogni aggiornamento riduce il più possibile lungo la direzione scelta. Questo garantisce che il metodo scenda più velocemente possibile in quella direzione.

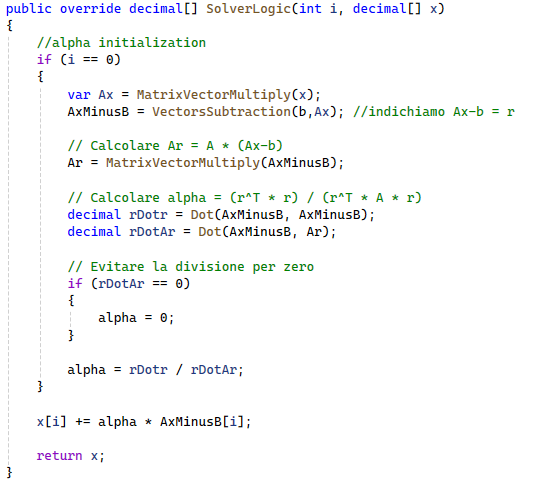
Se Consideriamo , l'i-esimo elemento del vettore . La formula di aggiornamento può essere scritta come:

Se andiamo a visualizzare quindi l’implementazione del metodo SolverLogic all’interno della classe GradientSolver possiamo vedere che implementerà esattamente la formula di aggiornamento di ogni componente appena descritta:

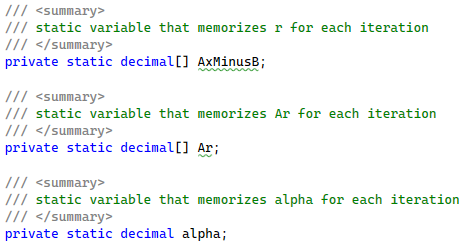
Calcolo del residuo:

Calcolo di alpha:

Aggiornamento di , ossia:



Nota che per tenere traccia dei valori del residuo e di alpha si definiscoefini questi come attributi delle classe, in modo che ogni chiamata del metodo SolverLogic per condivida gli stessi valori. Solo quando si inizia a calcolare un nuovo vettore (ce se ne accorge perché i==0) allora il valore viene sovrascritto:



Siccome poi non abbiamo alcuna operazione aggiuntiva da fare per la prossima iterazione il metodo SolverExitCondition sarà identico al precedente, ossia:



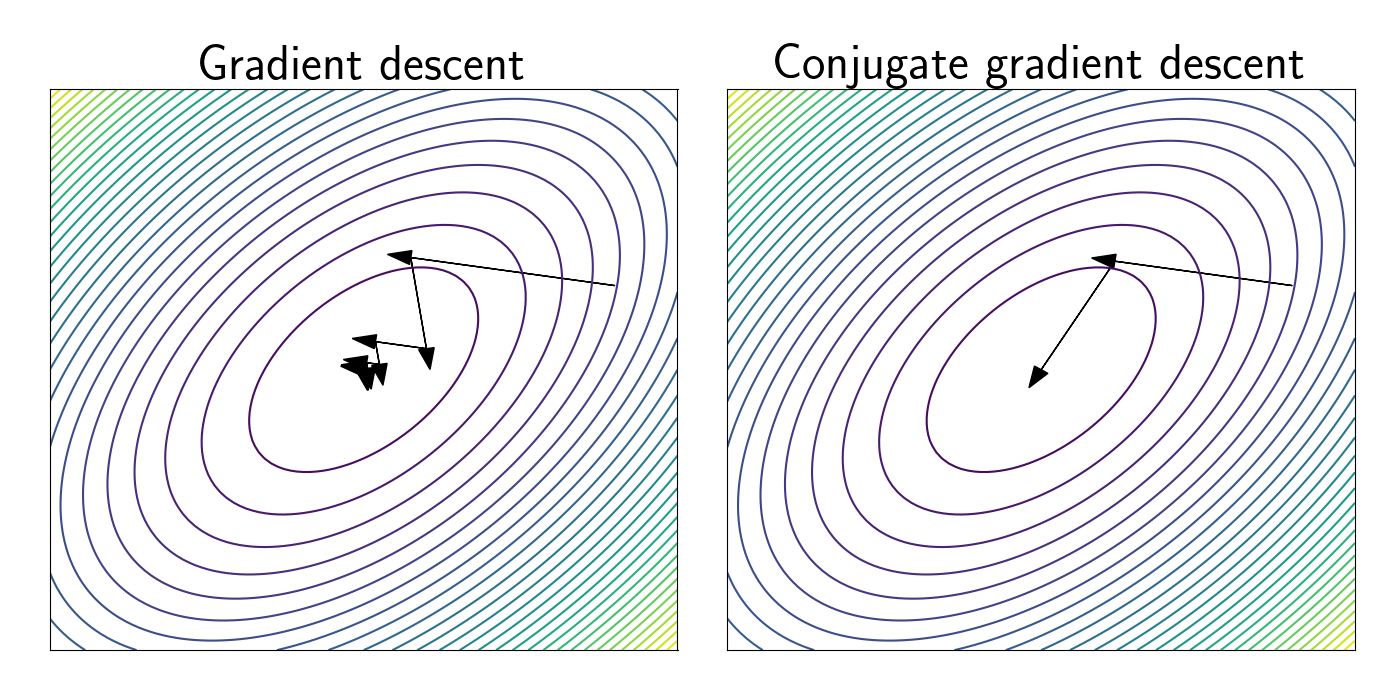
## 1.6. Implementazione GradientSolver

Il metodo del gradiente coniugato è un miglioramento del metodo del gradiente, che evita la convergenza a zig-zag tramite l’utilizzo di una “guida” chiamata positive. **passo**. Questo metodo garantisce la convergenza in un numero finito di passi per matrici simmetriche e definite positive. Anche qui quindi abbiamo che la funzione che a funzione da minimizzare è:

Il minimo di corrisponde alla soluzione del sistema . Mentre la funzione di aggiornamento è:

Dove:

* (nota che è calcolato con al posto che )
* è il passo del gradiente (vettore derivato da ) che serve ad indirizzare più precisamente verso la soluzione del sistema



Più nello specifico, per arrivare risolvere il problema di minimizzazione, abbiamo i seguenti steps:

1. **Inizializzazione di residuo e passo:** Si inizia con una stima iniziale del vettore delle incognite ​. Con questo si calcola:
   1. il residuo:
   2. Il passo:
2. **Aggiornamento di** : Una volta calcolati e è possibile calcolare alpha:ento del

ed avendo alpha e a è possibile calcolare l’aggiornamento del vettore delle incognite:

1. **Controllo convergenza** : Si controlla se la soluzione calcolata converge:
2. **Aggiornamento delle variabili per la prossima iterazione**: Abbiamo che dopo aver verificato se il vettore permette la convergenza, in caso negativo allora aggiorniamo le variabili per la prossima istanza:
   1. Residuo: che è dimostrabile essere un modo più efficiente di calcolare
   2. Beta: è un coefficiente di supporto per il calcolo di .
   3. Passo:

In ogni iterazione andremo a ripetere tutte le operazioni dalla 2 a alla 4.

Nota che se consideriamo , l'i-esimo elemento del vettore . La formula di aggiornamento può essere scritta come:

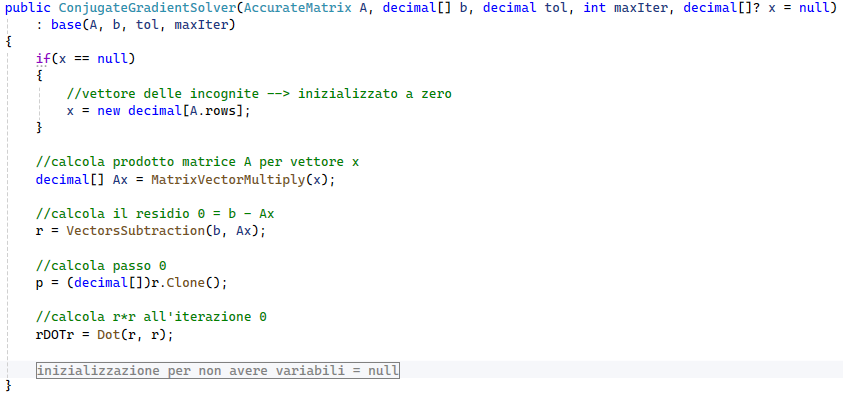
Diversamente dalle implementazioni precedenti, per realizzare questa serie di step in linguaggio C#, oltre che ad utilizzare l’implementazione del metodo SolverLogic all’interno della classe GradientSolver dovremo anche fare utilizzo del metodo costruttore della classe (in quanto, come abbiamo visto dal comportamento appena descritto, la parte iterativa del codice consisterà dallo step 2 allo step 4; lo step 1 è puramente pensato per il setup iniziale)

Calcolo di in quanto è una variabile utile sia per il calcolo di alpha che di beta, non avrebbe senso calcolarla 2 volte più tardi

Calcolo del primo passo:

Calcolo dealprimo coil resduo:

Se non viene fornito il vettore delle incognite allora viene inizializzato a 0



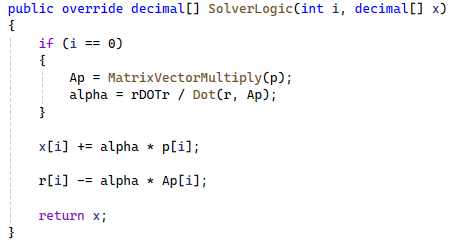
Una volta fatto il setup delle variabili (corrispondente allo step 1 descritto sopra) possiamo vedere come il metodo metodo SolverLogic (all’interno della classe GradientSolver) implementi esattamente lo step 2 (formula di aggiornamento di ogni componente ) appena descritta:

Aggiornamento di

Calcolo di alpha:

Nota che questa operazione viene eseguita una sola volta per ogni

Aggiornamento di



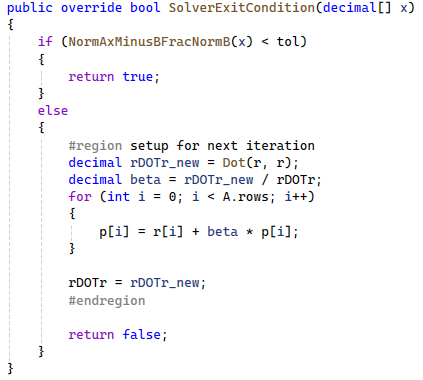
Nota che nella descrizione che abbiamo dato prima l’aggiornamento del residuo sarebbe da fare dopo il controllo della convergenza, noi in questo caso lo calcoliamo prima in quanto nel metodo calcconveraggiornadescrSolverExitCondition (per scelta di design) non viene passato il parametro quindi non sarebbe possibile calcolare ma andrebbe calcolato ad ogni iterazione tutto (che sarebbe un grosso spreco di risorse computazionali)

In fine, il metodo SolverExitCondition sarà incaricato, in questo caso, di completare lo step 3 e lo step 4:

**Step 4**: In caso di non convergenza si preparano le variabili per l’iterazione successiva, abbiamo infatti:

* Calcolo di beta:
* Aggiornamento del passo: Aggiornam
* Aggiornamento della variabile rDOTrOTvariabggiorna a teraprossi)

**Step 3**: Controllo della convergenza , si controlla checontrol



Come si può notare, questo è il metodo che ha reso necessario renderenece SolverExitCondition un metodo astratto, in quanto questo metodo aveva bisogno di eseguire delle operazioni di setup per la prossima iterazione caso in cui non si raggiungesse la convergenzaaastratt.

|  |  |
| --- | --- |
|  | VanArsdel, Ltd. |
| Taylor Phillips  5678 Main St  New York, NY 90210  September 16, 20XX |  |

|  |
| --- |
|  |
| Dear Taylor Phillips,  Write the body of your letter here. To update any of the letter's information, select the text, and start typing.  Want to change fonts? Go to the Home tab and choose Fonts. You can use a built-in font combination or select one of your own.  You can also change the colors of the template to match your personal taste. Go to the Design tab and choose a color palette from the Colors menu. Hovering over the different palettes will show you what your document would look like with the new palette.  To change the color or font formatting back to the original settings, go to the Design tab, and select the Theme menu. From there, choose the option to reset the original template theme.  Warm Regards,  Jordan Mitchell  CEO  5678 Main St.  New York, NY 90210  212-555-0199 |
| www.vanarsdelltd.com | jordan@vanarsdelltd.com |

|  |  |
| --- | --- |
|  | VanArsdel, Ltd. |
|  | 5678 Main St.  New York, NY 90210  212-555-0199 |
| Taylor Phillips  5678 Main St  New York, NY 90210  September 16, 20XX |  |

|  |
| --- |
|  |
| Dear Taylor Phillips,  Write the body of your letter here. To update any of the letter's information, select the text, and start typing.  Want to change fonts? Go to the Home tab and choose Fonts. You can use a built-in font combination or select one of your own.  You can also change the colors of the template to match your personal taste. Go to the Design tab and choose a color palette from the Colors menu. Hovering over the different palettes will show you what your document would look like with the new palette.  To change the color or font formatting back to the original settings, go to the Design tab, and select the Theme menu. From there, choose the option to reset the original template theme.  Warm Regards,  Jordan Mitchell  CEO |
| www.vanarsdelltd.com | jordan@vanarsdelltd.com |