Laboratorio di Programmazione e Calcolo
Canale 2
Appunti del corso

Simone Cacace e Giuseppe Visconti
Dipartimento di Matematica
Sapienza Università di Roma

Anno Accademico 2025–2026

9 Visualizzazione dati: `Gnuplot`

Nel contesto del calcolo scientifico, la visualizzazione grafica dei dati gioca un ruolo fondamentale per interpretare risultati numerici, analizzare l'andamento di funzioni e confrontare modelli teorici con osservazioni sperimentali.

Il linguaggio `C` non ha strumenti nativi per la grafica: per visualizzare i risultati, è necessario utilizzare librerie dedicate, oppure esportare i dati su file e utilizzare un software esterno per il tracciamento.

In questo corso forniremo le nozioni base per l'utilizzo di `Gnuplot`, un potente strumento per la visualizzazione grafica di dati e funzioni matematiche.

`Gnuplot` è un programma open-source per la generazione di grafici 2D e 3D. È leggero, multipiattaforma e particolarmente adatto per l'integrazione con linguaggi di programmazione come `C`, `Python` o `Fortran`.

Nato nel 1986, è ampiamente usato in ambito scientifico per tracciare grafici di funzioni analitiche o numeriche, rappresentare dati sperimentali e per creare visualizzazioni per articoli, relazioni e presentazioni.

In particolare, useremo `Gnuplot` per tracciare grafici di funzioni calcolate numericamente, visualizzare l'evoluzione temporale delle variabili di un sistema dinamico, confrontare soluzioni esatte e approssimate.

Esistono tre modi principali per usare `Gnuplot` nel nostro contesto:

- Uso interattivo: si lancia `Gnuplot` da terminale e si scrivono comandi direttamente.
- Script: si prepara un file di comandi che `Gnuplot` esegue in batch.
- Chiamate da `C`: si mandano comandi a `Gnuplot` direttamente dal programma `C` in fase di esecuzione.

9.1 Installazione di `Gnuplot`

Linux

Nella maggior parte delle distribuzioni Linux `gnuplot` è già disponibile nei repository ufficiali. Aprire un terminale e digitare:

```
sudo apt install gnuplot
```

Per verificare l'installazione:

```
gnuplot --version
```

macOS

Su macOS è possibile installare `gnuplot` tramite il gestore di pacchetti `Homebrew`. Se `brew` non è installato, si può aggiungere seguendo le istruzioni su `https://brew.sh`. Successivamente:

```
brew install gnuplot
```

Per verificare:

```
gnuplot --version
```

Windows

Su Windows, il modo più semplice è scaricare l'eseguibile dal sito:
https://sourceforge.net/projects/gnuplot/files/gnuplot/6.0.3/

Scaricare il file di installazione `gp603-win64-mingw.exe`. Durante l'installazione, lasciare attiva l'opzione "Add gnuplot to PATH" in modo da poterlo eseguire dal prompt dei comandi. Altra nota importante: in fase di installazione scegliere i terminali qt e wxt.

Per verificare l'installazione, aprire il terminale `cmd` o `PowerShell` e digitare:

```
gnuplot --version
```

9.2 Avvio di Gnuplot

Per avviare gnuplot aprire un terminale e digitare il comando

```
gnuplot
```

Si noti come il prompt dei comandi cambia indicando che gnuplot è attivo

```
  G N U P L O T
  Version 5.4 patchlevel 2    last modified 2021-06-01

  Copyright (C) 1986-1993, 1998, 2004, 2007-2021
  Thomas Williams, Colin Kelley and many others

  gnuplot home:     http://www.gnuplot.info
  faq, bugs, etc:   type "help FAQ"
  immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'wxt'
gnuplot>
```

A questo punto gnuplot è pronto a ricevere comandi. Per uscire usare il comando

```
exit
```

oppure

```
quit
```

oppure la sequenza ctrl-d.

Variabile d'ambiente

Affinché il tutto funzioni correttamente, `gnuplot` deve essere un comando disponibile globalmente nel sistema, ovvero accessibile via terminale da qualunque posizione nell'albero delle cartelle. In ambiente Linux e Mac questo è il default di installazione per molti programmi,

sotto Windows potrebbe essere necessario inserire Gnuplot nel cosidetto PATH di sistema. Per verificare:

- Aprire cmd (Prompt dei comandi).
- Digitare

```
gnuplot --version
```

- Se nel terminale viene riportato il numero di versione, ad esempio

```
gnuplot 5.4 patchlevel 2
```

  allora `gnuplot` è disponibile globalmente.

- Se non è riconosciuto occorre andare in:
  `Pannello di controllo -> Sistema -> Impostazioni di sistema avanzate -> Variabili d'ambiente`
  e modificare la variabile `PATH` nelle variabili di sistema aggiungendo il percorso, ad esempio:
  `C:\Program Files\gnuplot\bin`

- Riavviare il prompt dei comandi.

9.3 Uso interattivo di `Gnuplot`

9.3.1 Funzioni di una variabile

Il comando principale di `gnuplot` è sicuramente `plot`, che può essere usato in molti modi differenti a seconda dei parametri passati in input.

Il modo più semplice consiste nel plottare il grafico cartesiano di una funzione di una variabile, definita da una espressione analitica. Ad esempio:

```
plot sin(x)
```

La variabile `x` è una variabile riservata da `gnuplot` proprio al plot di grafici di funzioni della variabile `x` in una dimensione.

Concatenando e componendo funzioni elementari è possibile plottare grafici più complicati. Ad esempio:

```
plot x*cos(log(1+abs(x**2)))*exp(-sin(x))
```

Le operazioni e funzioni matematiche implementate in `gnuplot` sono di base le stesse della libreria `C <math.h>`, con qualche eccezione, ad esempio l'elevamento a potenza che in gnuplot si ottiene tramite l'operatore `**`.

Possiamo anche specificare lo spessore di linea con l'opzione `linewidth`. Ad esempio:

```
plot sin(x) linewidth 2
```

Oppure, dopo aver eseguito il `plot`, con il comando `set` e opzione `linetype`:

```
set linetype 1 linewidth 2
```

dove `linetype 1` indica che vogliamo cambiare la prima linea che compare in figura.

Quando si usa l'opzione `set` diventa necessario forzare l'aggiornamento del grafico con

```
replot
```

che ripete l'ultimo comando plot con le preferenze definite da `set`.

Dominio e codominio. I grafici finora prodotti hanno tutti per dominio di default l'intervallo $[-10, 10]$. Per cambiarlo utilizzare il comando

```
set xrange[xmin:xmax]
```

Ad esempio (la variabile `pi` è riservata e vale π):

```
set xrange[0:2*pi]
plot cos(x)
```

oppure dopo il `plot`

```
plot cos(x)
set xrange[0:2*pi]
replot
```

Il codominio per la variabile `y` (anch'essa riservata) viene per default impostato tra il valore minimo ed il valore massimo che la funzione plottata assume nel dominio (in questo caso tutti i valori in $[-1, 1]$).

Per cambiarlo utilizzare il comando

```
set yrange[ymin:ymax]
```

ricordando che, se la funzione assume valori al di fuori dal range impostato, non tutto il grafico sarà visibile.

Possiamo inserire anche un'etichetta agli assi:

```
set xlabel 'asse x'
set ylabel 'asse y'
```

Plot multipli. Per plottare più di un grafico alla volta è sufficiente elencare tutte le funzioni in un singolo comando `plot`, separando le varie espressioni con delle virgole.

Ad esempio:

```
plot cos(x), cos(2*x), cos(3*x), cos(4*x)
```

`Gnuplot` assegna ad ogni funzione un colore differente di default e riporta il nome ed il colore in una legenda (`key`) in alto a destra.

Si può cambiare la posizione della legenda, ad esempio per posizionarla in alto a sinistra:

```
set key left top
```

Di nuovo, possiamo modificare anche le proprietà delle varie linee:

```
set linetype 1 linewidth 3 linecolor rgb 'red'
set linetype 1 linewidth 3 linecolor rgb 'blue'
set linetype 1 linewidth 3 linecolor rgb 'green'
```

```
set linetype 1 linewidth 3 linecolor rgb 'cyan'
replot
```

9.3.2 Curve parametriche nel piano

Una curva nel piano può essere definita tramite una coppia di equazioni parametriche del tipo $(x(t), y(t))$, dove x e y sono funzioni reali del parametro t che varia in un intervallo $[t_0, t_1]$.

Per plottare curve in gnuplot occorre passare alla modalità parametrica utilizzando il comando

```
set parametric
```

Da questo momento in poi la variabile riservata per i grafici non è più `x` ma `t`.

Un semplice esempio di curva nel piano è la circonferenza di raggio unitario, che può essere parametrizzata tramite la coppia di equazioni $(cos(t), sin(t))$, dove il parametro t varia in $[0, 2\pi]$.

Per impostare il dominio in `t` della curva, utilizzare il comando (analogo a xrange)

```
set trange[0:2*pi]
```

Per plottare quindi la curva occorre passare al comando plot le due equazioni parametriche separate da una virgola:

```
plot cos(t), sin(t)
```

In modalità parametrica questa espressione viene interpretata come una singola curva e non come due funzioni diverse. Inoltre in modalità non parametrica si otterrebbe un errore perché la variabile riservata dovrebbe essere `x` e non `t`.

Il risultato precedente sembra mostrare un'ellisse e non una circonferenza. Questo dipende dal fatto che gli assi cartesiani non sono scalati di default in maniera uniforme. D'altra parte la "circonferenza" è tangente ai bordi della figura, proprio perché `xrange` e `yrange` sono impostati di default per contenere tutto il plot. Per risolvere il problema della scalatura non uniforme utilizzare il comando:

```
set size square
replot
```

Per cambiare invece l'estensione della figura (bounding box) si utilizzano, come visto precedentemente, i comandi:

```
set xrange[-2:2]
set yrange[-2:2]
replot
```

che in questo contesto parametrico corrispondono entrambi al codominio.

Per plottare più di una curva parametrica alla volta è sufficiente elencare tutte le coppie di funzioni in un singolo comando `plot`, separando le varie espressioni con delle virgole. Ad esempio il comando

```
plot cos(t), sin(t) , 1.5*cos(t), 1.5*sin(t)
```

disegna due circonferenze, rispettivamente di raggio 1 e 1.5.

Attenzione: Notare il ruolo differente tra le virgole che definiscono le coppie di equazioni parametriche e la virgola che separa i grafici delle due curve.

Per tornare in modalità cartesiana digitare il comando

```
unset parametric
```

9.3.3 Curve parametriche nello spazio

Una curva nello spazio può essere definita tramite una terna di equazioni parametriche del tipo $(x(u), y(u), z(u))$, dove x, y e z sono funzioni del parametro u che varia in un intervallo $[u_0, u_1]$.

A differenza delle curve nel piano, per le curve nello spazio la variabile riservata è `u` e non `t`.

Ricordare sempre di passare in modalità parametrica quando si vuole disegnare questo tipo di grafici

```
set parametric
```

Un esempio: disegnare un'elica che compia 4 giri completi intorno all'origine, con un raggio che decresca da 1 e 0 e che abbia un'altezza complessiva di 2.

Impostare dunque il dominio della curva come richiesto, tramite il comando

```
set urange[0:8*pi]
```

e procedere con il comando `splot`, versione di `plot` per oggetti 3D:

```
splot (1-u/(8*pi))*cos(u) , (1-u/(8*pi))*sin(u) , u/(4*pi)
```

9.3.4 Lettura di dati da file

Tipicamente i grafici che si vogliono visualizzare sono il risultato di misurazioni o di simulazioni numeriche. Difficilmente (praticamente mai!) si ha a disposizione l'espressione analitica di una funzione che descriva correttamente un fenomeno o rappresenti la soluzione di un problema, ma solo dati più o meno strutturati.

`Gnuplot` è in grado di leggere e visualizzare dati da file, purché vengano rispettate alcune regole di base e purché i dati siano organizzati in maniera opportuna.

Conviene sempre raccogliere i propri file di dati in cartelle apposite e assicurarsi di lanciare `gnuplot` direttamente dalla cartella di interesse. In alternativa si può usare da `gnuplot` il comando

```
cd "path/to/folder"
```

dove la stringa `path/to/folder` identifica una posizione specifica nell'albero delle cartelle.

File a colonna singola. Creare un file di testo denominato `dati1.txt` contenente alcuni valori scritti in colonna. Dal terminale è possibile visualizzare il contenuto di un file denominato `nome_file.txt` con il comando

```
cat nome_file.txt
```

Supponiamo che il nostro file di testo `dati1.txt` contenga i seguenti dati

```
# dati1.txt - Temperature a intervalli di un secondo
36.5
36
36.5
37
38
38.5
37.5
36.5
36
36.5
```

che rappresentano, ad esempio, diverse misurazioni di temperatura all'interno di una stanza.

Ogni occorrenza del carattere cancelletto # nel file, quando letto da `gnuplot`, viene interpretato come l'inizio di un commento e pertanto viene ignorato in fase di caricamento insieme a tutta la riga che lo segue.

Per default `Gnuplot` interpreta i dati di un file di questo tipo come come punti del piano xy in cui l'ascissa è uguale all'indice di riga del record letto (partendo da 0), mentre l'ordinata è il valore del record letto.

Nel caso del file `dati1.txt`, le coppie di coordinate generate saranno quindi

$$(0, 36.5), (1, 36), (2, 36.5), (3, 37), \ldots$$

Per plottare i dati di un file utilizzare il comando base

```
plot 'nomefile'
```

Come già detto precedentemente, `Gnuplot` modifica automaticamente il bounding box del grafico per contenere tutti i dati plottati. Poiché i punti che cadono sui bordi della figura potrebbero non essere ben visibili, occorre eventualmente regolare `xrange` e `yrange` prima di visualizzare il grafico:

```
set xrange[-1:10]
set yrange[35:41]
plot 'dati1.txt'
```

A differenza del plot di funzioni analitiche, `Gnuplot` disegna i dati come singoli campioni (markers), utilizzando alcune forme grafiche di default (+ , x , * , o , ...).

Possiamo modificare le proprietà grafiche dei markers sia durante il comandp `plot` che dopo usando, come già visto, il comando `set` con opzione `linetype`. Ad esempio:

```
  set linetype 1 pointtype 4 pointsize 3
  replot
```

dove `pointtype` permette di cambiare stile del marker, mentre `pointsize` permette di modificare dimensione del marker.

Se invece si desidera che ogni dato sia unito al successivo con una linea (ovvero che i dati vengano interpolati linearmente) si può usare il comando

```
plot 'dati1.txt' with lines
```

File a più colonne. Come nel caso di funzioni analitiche, per plottare più di un grafico di dati alla volta è sufficiente elencare tutti i file in un singolo comando `plot`, separando le varie espressioni con delle virgole. Ad esempio,

```
plot 'dati1a.txt' with lines, 'dati1b.txt' with lines
```

Tuttavia, se i dati sono coerenti (stesso numero) e rapresentano due o più misurazioni di uno stesso fenomeno, può essere conveniente utilizzare un solo file, memorizzando più colonne di valori, una per ogni misurazione.

Creare un file di testo denominato `dati2.txt` contenente due colonne di valori

```
#dati2.txt - Due temperature a intervalli di un secondo
36.5 38
36 37
36.5 36
37 39
38 37.5
38.5 36.5
37.5 37
36.5 38
36 39
36.5 36
```

Attenzione: i record di ogni riga devono essere separati da almeno uno spazio!

Affinché `Gnuplot` generi due sequenze di coppie di coordinate, rispettivamente

$$(0,36.5),(1,36),(2,36.5),(3,37),\ldots$$

e

$$(0,38),(1,37),(2,36),(3,39),\ldots$$

occorre utilizzare il parametro `using` seguito dal numero della colonna da cui leggere i dati (partendo da 1). Ad esempio:

```
plot 'dati2.txt' using 1 with lines, 'dati2.txt' using 2 with lines
```

oppure

```
plot 'dati2.txt' using 1 with lines, '' using 2 with lines
```

dove i due apici singoli '' sono una abbreviazione per "l'ultimo file caricato" (in questo caso `dati2.txt`).

Si supponga ora che il file di dati contenga coordinate di punti nel piano cartesiano, ovvero coppie del tipo $(x_0,y_0),(x_1,y_1),(x_2,y_2),\ldots$. Creare un file di testo denominato `dati3.txt` come segue, separando le coordinate con (almeno) uno spazio:

```
#dati3.txt - Coppie di coordinate nel piano
0 0
0.1 0.5
0.25 0.7
0.35 1
0.5 2
0.7 0
1 1
1.3 2.7
1.5 3
2 1
```

Si osservi che la struttura del file `dati3.txt` è identica a quella del file `dati2.txt`, ma ora le coppie di valori rappresentano dati 2D e non fanno più riferimento a diverse misurazioni di un dato 1D (uno per colonna). Se dunque non viene specificato il parametro di colonna using, `Gnuplot` interpreta di default i dati come coordinate cartesiane, come si può verificare utilizzando i comandi

```
set xrange[-1:3]
set yrange[-1:4]
plot 'dati3.txt' with lines
```

Creare un file di testo denominato `dati4.txt` con la seguente struttura

```
#dati4.txt - Terne di coordinate nel piano
x0 y10 y20
x1 y11 y21
x2 y12 y20
...
```

In questo caso l'utilizzo del parametro using permette di generare le due sequenze di coordinate, specificando le colonne da cui prendere i dati

$$(x0, y10), (x1, y11), (x2, y12), \ldots$$

e

$$(x0, y20), (x1, y21), (x2, y22), \ldots$$

tramite il comando

```
plot 'dati4.txt' using 1:2 with lines, '' using 1:3 with lines
```

ovvero i primo grafico con dati letti dalle colonne 1:2 e il secondo dalle colonne 1:3.

9.3.5 Abbreviazioni

In `Gnuplot` ogni comando o parametro può essere abbreviato (in alcuni casi anche con una singola lettera), purché non ci sia ambiguità con un altro comando o parametro (in tal caso viene segnalato un errore).

`Esercizio:` investire sulla pigrizia, trovare il numero minimo di lettere per scrivere tutti i comandi studiati finora.

Un esempio:

```
se xr[0:1]
se yr[0:1]
p x**2 w l
p 'dati.txt' u 1:3 w l, '' u 1:4 w l
```

9.4 Script

Oltre alla modalità interattiva discussa finora, `Gnuplot` permette l'esecuzione di script.

Uno script è semplicemente un normale file di testo (cui si da solitamente estensione .gnp per distinguerlo) contenente sequenze di comandi `Gnuplot`. Questo permette di riutilizzare codice scritto precedentemente.

Creare un file denominato `script1.gnp` contenente i seguenti comandi

```
set xrange[-2:2]
plot 1/(1+x**2) with lines
```

A questo punto per eseguire lo script si può utilizzare, direttamente da `Gnuplot`, il comando

```
load 'script1.gnp'
```

In alternativa si può eseguire uno script da terminale, senza prima aver lanciato `Gnuplot`. In tal caso si utilizza il comando

```
gnuplot script1.gnp
```

Attenzione: Il file non contiene apici, lo script diventa un parametro di input per `gnuplot`.

Tuttavia si osserva che il comando non produce alcun effetto, oppure la finestra grafica appare per un breve istante prima di chiudersi. Questo perché si sta chiedendo al sistema operativo di aprire `Gnuplot`, caricare lo script e poi uscire.

Per ovviare al problema si può utilizzare il comando

```
gnuplot script1.gnp -persist
```

che permette di lasciare aperta (persistente) la finestra grafica generata da `Gnuplot` prima di restituire il prompt dei comandi nel terminale.

Trattandosi di un normale file di testo, uno script di comandi `Gnuplot` può essere anche generato da un codice `C`. Sarà sufficiente aprire un file con fopen e scrivere i comandi `Gnuplot` utilizzando `fprintf`, rispettando rigorosamente la sintassi di `Gnuplot`, altrimenti ci sarà un errore di interpretazione al lancio dello script.

Ad esempio, se si vuole inserire nello script il comando `Gnuplot`

```
plot sinh(x) with lines
```

nel corrispondente codice `C` che lo genera si utilizzerà (eventualmente abbreviando i comandi)

```
fprintf(fp,"p sinh(x) w l\n");
```

Una volta generato lo script `Gnuplot` tramite codice `C`, l'esecuzione procede come discusso precedentemente. In alternativa si può lanciare direttamente lo script dal codice `C`, utilizzando la funzione system presente nella liberia `<stdlib.h>`. E' sufficiente chiamare la funzione subito dopo la creazione dello script, passando il comando che si vuole eseguire:

```
system("gnuplot script1.gnp");
```

oppure

```
system("gnuplot script1.gnp -persist");
```

9.5 Pipe da C a Gnuplot

Talvolta, all'interno di un codice `C`, può essere utile disegnare un grafico al volo, senza necessariamente salvare su disco i dati corrispondenti e poi generare uno script `gnuplot` che li visualizzi.

A tale scopo, `Gnuplot` consente una modalità manuale di immissione dati. Si procede esattamente come nel plot di file di dati, ma si utilizza il simbolo `'-'` , un "nome speciale" di file che attiva la modalità manuale. Una volta digitato il comando

```
plot '-' w l
```

e premuto invio, gnuplot risponde con

```
input data ('e' ends) >
```

attendendo l'inserimento di coordinate di punti, separate da spazi e seguite da un invio. In questo caso si inseriscono solo coppie di coordinate perché si è utilizzato `plot`, in modo analogo si può usare `splot` inserendo terne di coordinate. Una volta terminato l'inserimento si digita `e` seguito da un uteriore invio e viene così generato il grafico corrispondente.

Chiaramente questa modalità manuale non è pensata per grosse moli di dati, ma è la chiave per realizzare un plot immediato dal `C`. Infatti, da un codice `C`, è possibile inviare dati ad un altro programma che li accetta in input (esattamente quello che si vuole fare!), tramite un un meccanismo noto come pipe (tubo). La sintassi è identica a quella di scrittura su file, ma anziché usare `fopen` e `fclose` si utilizzano le funzioni `popen` e `pclose` (rispettivamente pipe open e pipe close). Il "file" da aprire in "scrittura" è esattamente il programma a cui si vuole inviare dati, in questo caso proprio `Gnuplot`:

```
FILE *pf;
pf=popen("gnuplot -persist","w");
```

A questo punto `Gnuplot`, che inizia a girare in un altro processo, è pronto a ricevere comandi, esattamente come nella modalità immediata. Si possono quindi inviare comandi per impostare parametri, utilizzando anche variabili presenti nel codice `C`, eventualmente frutto di un qualche calcolo numerico. Ad esempio

```
#define N 100
double xmin=-1, xmax=1;
double x[N],y[N];
...
fprintf(pf,"set xrange[%lf:%lf]\n",xmin,xmax);
...
```

Si invia dunque a `Gnuplot` il comando di plot in modalità manuale e si procede a passare i dati, esattamente come se li si stesse immettendo tramite la tastiera (notare infatti la presenza dei caratteri `\n` per andare a capo in tutte le occorrenze di `fprintf`):

```
fprintf(pf,"plot '-' with lines\n");
for(int i=0;i<N;i++){
  fprintf(pf,"%lf %lf\n",x[i],y[i]);
}
fprintf(pf,"e\n");
fflush(pf);
```

Una volta inviato il carattere `'e'`, `Gnuplot` realizza il grafico. In caso di errori di sintassi `Gnuplot` risponde come in modalità immediata, ma il pipe `C` non si arresta, per cui tutti i comandi successivi al primo errore continueranno a produrre errori.

La funzione `fflush(pf)` serve a forzare lo svuotamento di un buffer di output, ad esempio per garantire che tutti i dati siano trasferiti prima di un nuovo invio (si pensi a un loop di animazione in cui ogni fotogramma corrisponde a un set di dati diversi).

Si conclude chiudendo il pipe

```
pclose(pf);
```

e il codice C procede fino alla fine del main. Si osservi che la finestra grafica di Gnuplot rimane aperta dopo l'esecuzione del codice C, dal momento che Gnuplot è stato aperto in questo esempio col parametro -persist.

Il seguente codice C riassume i concetti finora esposti, generando al volo in gnuplot l'animazione (5 secondi) di una circonferenza pulsante!

```
#include <stdio.h>
#include <math.h>

int main(){

 double xmin=-2, xmax=2, ymin=-2, ymax=2;
 double smin=0, smax=2*M_PI, ds;
 double radius, t=0, dt=0.01, tmax=5;
 int Ns=100;

 ds=(smax-smin)/(Ns-1);

 FILE *pf;
 pf=popen("gnuplot","w");
 fprintf(pf,"set xrange[%lf:%lf]\n",xmin,xmax);
 fprintf(pf,"set yrange[%lf:%lf]\n",ymin,ymax);
 fprintf(pf,"set size square\n");
 fprintf(pf,"unset key\n");
 // linux/mac
 fprintf(pf,"set terminal x11\n");
 // windows: provare fprintf(pf,"set terminal windows\n");

 while (t<tmax){
  radius=(0.5+0.5*(1+cos(2*M_PI*t)));
  fprintf(pf,"set title 'Time = %.2lf'\n",t);
  fprintf(pf,"plot '-' w l\n");
  for(int s=0;s<Ns;s++){
   fprintf(pf,"%lf %lf\n",radius*cos(s*ds),radius*sin(s*ds));
  }
  fprintf(pf,"e\n");
  fprintf(pf, "pause %lf\n",dt);
  fflush(pf);
  t+=dt;
 }
 pclose(pf);

 return 0;
}
```

Laboratorio di Programmazione e Calcolo
Canale 2
Appunti del corso

Simone Cacace e Giuseppe Visconti
Dipartimento di Matematica
Sapienza Università di Roma

Anno Accademico 2025–2026

12 Equazioni non lineari

Il problema di cui ci occuperemo in questo capitolo è quello della ricerca degli zeri di funzioni reali. Considereremo il caso scalare, per cui il problema si può riscrivere come: assegnata una funzione $f : \mathbb{R} \to \mathbb{R}$, continua su un intervallo $[a, b]$, si cerca $x \in [a, b]$ tale che $f(x) = 0$.

Raramente le radici di f sono note in forma esplicita. In generale si cerca una successione $\{x_k\}$ tale che $\lim_{x_k \to \infty} x_k = \alpha$, dove α è una delle soluzioni del problema.

Teorema 1 (degli zeri per funzioni continue)**.** *Data una funzione $f : [a, b] \to \mathbb{R}$, continua in $[a, b]$ e tale che $f(a)f(b) < 0$, allora esiste $\alpha \in]a, b[$ tale che $f(\alpha) = 0$.*

Osservazione. Notare che la condizione $f(a)f(b) < 0$ garantisce l'esistenza di almeno uno zero nell'intervallo, ma potrebbero essercene più di uno; inoltre possono presentarsi radici multiple. Nel seguito ci limiteremo al caso di radici semplici.

12.1 Il metodo di bisezione

Il metodo di bisezione richiede solo che f sia continua, e prende spunto dal teorema degli zeri prima enunciato.

Tale metodo costruisce una successione di intervalli $[a_k, b_k]$ che contengono la radice α e il cui diametro tende a zero. Si parte da $[a_0, b_0] = [a, b]$. Ad ogni passo si calcola il punto medio

$$c_k = \frac{a_k + b_k}{2},$$

e si sceglie l'intervallo successivo come

$$[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, c_k], & \text{se } f(a_k)f(c_k) < 0, \\ [c_k, b_k], & \text{altrimenti.} \end{cases}$$

Si continua ripetendo il procedimento nel semi-intervallo selezionato (quello che contiene la radice), quindi si itera fino a quando l'intervallino sia sufficientemente piccolo. A questo punto basterà prendere il suo punto medio come approssimazione della radice α.

Errore e stima sulle iterazioni. La lunghezza dell'intervallo dopo k passi è

$$b_k - a_k = \frac{b_{k-1} - a_{k-1}}{2} = \cdots = \frac{b - a}{2^k}.$$

Poiché c_k è il centro dell'intervallo $[a_k, b_k]$, l'errore al passo k soddisfa

$$e_k := |c_k - \alpha| \leq \frac{b_k - a_k}{2} = \frac{b - a}{2^{k+1}} \longrightarrow 0 \qquad (k \to \infty).$$

Da questa relazione si ricava facilmente quante iterazioni sono necessarie per raggiungere una tolleranza prefissata Tol. Infatti, è sufficiente

$$b_k - a_k = \frac{b - a}{2^k} \leq \text{Tol} \qquad \Longrightarrow \qquad k \geq \left\lceil \log_2 \frac{b - a}{\text{Tol}} \right\rceil.$$

Un ragionamento utile riguarda il numero di iterazioni necessarie per guadagnare una cifra decimale di accuratezza. Poiché l'errore viene ridotto sempre dello stesso fattore (pari a $1/2$) ad ogni passo, ci chiediamo per quale k si ha

$$\frac{1}{2^k} \leq \frac{1}{10}.$$

Dal momento che $\log_2 10 \approx 3.32$, occorrono circa 3–4 iterazioni per ridurre l'errore di un fattore 10

Velocità di convergenza. Ricordiamo dal capitolo precedente il concetto di velocità di convergenza.

Sia $\{x_k\}$ una successione che converge a $\{\alpha\}$. La *velocità di convergenza* misura quanto rapidamente l'errore e_k tende a zero. Si dice che il metodo ha *convergenza lineare* se esiste una costante $M < 1$ tale che

$$e_{k+1} \leq M\, e_k \qquad \text{per } k \text{ sufficientemente grande.}$$

In questo caso l'errore si riduce di un fattore circa costante ad ogni iterazione. Si parla invece di *convergenza quadratica* quando

$$e_{k+1} \leq C\, e_k^2,$$

con una costante $C > 0$; in questo caso, una volta vicini alla soluzione, l'errore si abbatte molto più rapidamente.

Nel metodo di bisezione, poiché l'errore viene moltiplicato da $1/2$ ad ogni passo, il metodo di bisezione ha *convergenza lineare*. È quindi affidabile perché la convergenza è garantita, ma non molto veloce rispetto ad altri metodi iterativi.

Criteri di arresto. Il metodo di bisezione determina una successione, mediante iterazione, che converge alla soluzione del problema. Una domanda cruciale, come in qualsiasi altro metodo iterativo, è quando arrestare l'iterazione. Tipici test di arresto sono

residuo relativo:	$\lvert f(c_k)\rvert < C\,\mathrm{Tol},$
distanza tra estremi:	$\lvert b_k - a_k\rvert < \mathrm{Tol}\,\lvert c_k\rvert,$

dove C è una costante dell'ordine di f sull'intervallo (ad esempio $C = \max(|f(a)|, |f(b)|)$). Si può anche usare una combinazione dei due criteri.

12.2 Il metodo di punto fisso

Osserviamo che se $f, g : \mathbb{R} \to \mathbb{R}$ sono due funzioni legate dalla relazione $f(x) = x - g(x)$, allora si ha

$$f(\alpha) = 0 \quad \Longleftrightarrow \quad \alpha = g(\alpha).$$

In altre parole, α è uno zero di f se e solo se α è un *punto fisso* di g.

Una strategia per l'approssimazione dei punti fissi di g è la costruzione di una successione $\{x_n\}$ tale che

$$x_{n+1} = g(x_n), \quad n \geq 0,$$

con x_0 assegnato. Questa è la cosidetta *iterazione di punto fisso*.

Convergenza. Le domande sono: l'iterazione converge? E, se converge, converge effettivamente ad un punto fisso di g? Il seguente teorema garantisce ipotesi sufficienti per la convergenza delle iterazioni di punto fisso.

Teorema 2. *Data la successione $\{x_n\}$ definita da $x_{n+1} = g(x_n)$, $n \geq 0$, con x_0 assegnato. Se la funzione $g : [a, b] \to [a, b]$ è continua, derivabile e soddisfa*

$$|g'(x)| \leq M < 1, \quad \forall\, x \in [a, b],$$

Algorithm 1 Pseudocodice del metodo di bisezione per $f(x) = 0$

Require: Funzione f, estremi a, b tali che $f(a)f(b) \leq 0$, tolleranze Tol_x (per la lunghezza dell'intervallo), Tol_f (per il residuo), massimo iterazioni $k_{\max}$

if $f(a)\, f(b) > 0$ **then**
    return "Intervallo non valido: f(a)f(b) > 0"
end if
$k \leftarrow 0$
$c \leftarrow (a + b)/2$
while $k < k_{\max}$ **do**
    Calcola $c \leftarrow (a + b)/2$
    Calcola $f_c \leftarrow f(c)$
    (Criteri di arresto)
    if $|f_c| < \text{Tol}_f$ **then**
        print "Criterio su residuo raggiunto ad iterazione k"
        return c
    end if
    if $|b - a| < \text{Tol}_x\, |c|$ **then**
        print "Criterio su lunghezza intervallo raggiunto ad iterazione k"
        return c
    end if
    if $f(a)\, f_c \leq 0$ **then**
        $b \leftarrow c$
    else
        $a \leftarrow c$
    end if
    $k \leftarrow k + 1$
end while
print "Numero massimo di iterazioni raggiunto"
return c

allora g ha un unico punto fisso in $\alpha \in [a, b]$, e la successione $\{x_n\}$ converge ad α pur di scegliere $x_0 \in [a, b]$. Inoltre, se in aggiunta g' è continua in α

$$\lim_{n\to\infty} \frac{\alpha - x_{n+1}}{\alpha - x_n} = g'(\alpha),$$

ovvero la convergenza è lineare.

Dimostrazione. Le ipotesi del teorema assicurano che g è una contrazione in $[a, b]$, e pertanto ammette un unico punto fisso in $[a, b]$ per il Teorema del punto fisso di Banach.

Per il Teorema del valor medio si ha

$$\alpha - x_{n+1} = g(\alpha) - g(x_n) = g'(\xi_n)(\alpha - x_n), \quad \xi \in (\alpha, x_n).$$

Dunque

$$|\alpha - x_{n+1}| \leq M|\alpha - x_n| \leq M^{n+1}|\alpha - x_0| \to 0, \quad n \to \infty,$$

dato che $M^{n+1} \to 0$. Questo vuol dire che l'iterazione converge al punto fisso α. In particolare, si ha

$$\lim_{n\to\infty} \frac{\alpha - x_{n+1}}{\alpha - x_n} = g'(\alpha),$$

ovvero

$$|\alpha - x_{n+1}| \leq |g'(\alpha)||\alpha - x_n|,$$

che implica la convergenza lineare del metodo. □

Conseguenza del precedente teorema è che l'errore $e_n = |\alpha - x_n|$ soddisfa

$$e_{n+1} \approx |g'(\alpha)|e_n,$$

ovvero $|g'(\alpha)|$ rappresenta il fattore di riduzione dell'errore a convergenze. Pertanto, il metodo di punto fisso converge a velocità costante, tanto più velocemente quanto più $|g'(\alpha)|$ è piccolo. Possiamo allora definire *velocità di convergenza* dell'iterazione di punto fisso la quantità

$$R = \log \frac{1}{|g'(\alpha)|.}$$

Ricapitolando:

- se $|g'(\alpha)| < 1$ si possono avere due situazioni: convergenza monotona (rispettivamente non monotona) nel senso che $\alpha - x_n$ ha segno costante (rispettivamente alterno rispetto a n);
- se invece $|g'(\alpha)| > 1$, si possono avere le situazioni seguenti: la successione $\alpha - x_n$ diverge con segno costante se $g'(\alpha) > 1$, con segno oscillante se $g'(\alpha) < -1$.

Costruire g a partire da f. Un modo semplice per trasformare $f(x) = 0$ in un problema di punto fisso è scegliere

$$g(x) = x + \lambda f(x), \qquad \lambda \in \mathbb{R}.$$

Allora $g'(x) = 1 + \lambda f'(x)$. Per ottenere $|g'(x)| < 1$ occorre scegliere λ con segno opposto a $f'(x)$ e con modulo non troppo grande; in pratica si cerca λ tale che

$$|1 + \lambda f'(x)| < 1 \quad \text{per } x \text{ nell'intorno considerato.}$$

Esempio. Sia $f(x) = e^x(x^2 - 2)$ su $[1, 2]$, la radice è $\alpha = \sqrt{2}$. Qui $f'(x) = e^x(x^2 + 2x - 2) > 0$ su $[1, 2]$, dunque si può scegliere $\lambda < 0$ in $g(x) = x + \lambda f(x)$. Se si prende

$$|\lambda| < \frac{2}{\max_{x\in[1,2]} f'(x)} \approx \frac{2}{6e^2} \approx 0.045,$$

ad esempio $\lambda = -0.04$, l'iterazione di punto fisso converge localmente a $\sqrt{2}$.

Condizioni di arresto. Nel metodo di punto fisso l'iterazione deve essere arrestata quando si ritiene che l'approssimazione sia sufficientemente vicina al punto fisso α. Poiché la soluzione esatta non è nota, i criteri di arresto devono basarsi esclusivamente su quantità computabili, vale a dire gli iterati x_n o il residuo dell'equazione.

Il criterio più comune consiste nel richiedere

$$|x_{n+1} - x_n| \leq \text{Tol}.$$

ovvero si usa la differenza tra iterati come indicatore dell'errore vero.

Quando gli iterati possono assumere valori molto grandi o molto piccoli, è utile considerare anche la differenza relativa:

$$\frac{|x_{n+1} - x_n|}{|x_{n+1}|} \leq \text{Tol}.$$

Questo criterio evita che un errore assoluto piccolo risulti ingannevole se rapportato alla grandezza effettiva della soluzione.

Poiché il punto fisso α soddisfa $\alpha = g(\alpha)$, una misura naturale di quanto x_n sia vicino alla soluzione potrebbe essere anche il residuo

$$|g(x_n) - x_n|.$$

Si potrebbe quindi considerare come criterio di arresto

$$|g(x_n) - x_n| \leq \text{Tol}.$$

Un residuo piccolo indica che x_n quasi soddisfa l'equazione di punto fisso; tuttavia non sempre garantisce vicinanza alla soluzione. Infatti, dal Teorema del valor medio,

$$|g(x_n) - x_n| = |g'(\xi_n)|\,|\alpha - x_n|, \quad \xi_n \in (\alpha, x_n),$$

per cui, se $g'(\xi_n)$ è vicino a 1, il residuo può essere piccolo pur in presenza di un errore grande.

Indipendentemente dal criterio scelto, è sempre opportuno imporre anche un limite superiore al numero di iterazioni:

$$n < N_{\max},$$

in modo da evitare cicli infiniti nel caso in cui le ipotesi teoriche di convergenza non siano soddisfatte o il punto iniziale sia troppo distante dal punto fisso.

In pratica, una scelta robusta dei criteri di arresto combina:

$$\begin{cases} |x_{n+1} - x_n|/|x_{n+1}| \leq \text{Tol}, \\ n < N_{\max}. \end{cases}$$

Algorithm 2 Iterazioni di punto fisso $x = g(x)$

Require: Funzione g, punto iniziale x_0, tolleranza Tol_x (sul passo), massimo iterazioni $N_{\max}$

Calcola $x_1 \leftarrow g(x_0)$
$k \leftarrow 1$
if $|g(x_1) - x_1| < \text{Tol}_r$ **then**
    return x_1
end if
while $k < N_{\max}$ **do**
    Calcola $x_{k+1} \leftarrow g(x_k)$
    (Criteri di arresto)
    if $\dfrac{|x_{k+1} - x_k|}{|x_{k+1}|} \leq \text{Tol}_x$ **then**
        print "Criterio su iterate successive raggiunto ad iterazione k"
        return x_{k+1}
    end if
    $k \leftarrow k + 1$
end while
print "Numero massimo di iterazioni raggiunto"
return x_k

12.3 Metodi geometrici

I metodi geometrici usano più informazioni sulla funzione f (derivate o differenze incrementali) per accelerare la convergenza rispetto ai metodi di bisezione e punto fisso. Sono tipicamente più veloci ma non garantiti: la convergenza può fallire se la stima iniziale è lontana dalla radice.

12.3.1 Il metodo di Newton

Il metodo di Newton si basa sull'approssimazione locale di f con la retta tangente. Dato x_k, l'iterazione è

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{1}$$

Geometricamente ciò equivale a chiedere che x_{k+1} sia l'intersezione con l'asse delle ascisse della tangente alla curva $(x, f(x))$ nel punto $(x_k, f(x_k))$.

Dunque il metodo di Newton si può applicare per trovare uno zero α della funzione $f(x)$ purché in un intorno di α si abbia f differenziabile e $f'(x) \neq 0$.

Convergenza. A differenza del metodo di bisezione, che converge sempre purché la funzione cambi segno agli estremi dell'intervallo iniziale, la convergenza del metodo di Newton *non* è garantita per ogni scelta di x_0. Anzi, il metodo è in generale un metodo di *convergenza locale*: funziona solo se l'iterato iniziale x_0 è scelto sufficientemente vicino alla radice α.

Per comprendere questa limitazione, si osservi che la formula di Newton deriva dall'approssimazione lineare di f in un intorno di x_k:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k),$$

da cui si ottiene l'aggiornamento

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Questa approssimazione è accurata solo se x_k è abbastanza vicino alla radice; in caso contrario, la retta tangente può intersecare l'asse delle ascisse in un punto molto lontano, generando iterati che si allontanano rapidamente dalla soluzione oppure dando luogo a una successione oscillante o divergente.

Il risultato teorico fondamentale afferma che, se esiste una radice semplice α tale che $f(\alpha) = 0$ e $f'(\alpha) \neq 0$, e se f è almeno due volte derivabile in un intorno di α, allora esiste un intorno I della radice per cui, per ogni $x_0 \in I$, la successione di Newton è ben definita e converge alla radice ma solo a patto che x_0 sia scelto all'interno dell'intervallo di convergenza.

In pratica, questo significa che il metodo di Newton richiede una buona stima iniziale per essere efficace. L'intervallo di convergenza dipende in modo sensibile dalle caratteristiche locali della funzione: pendenze molto piccole o punti di flesso vicini alla radice possono rendere l'intervallo di convergenza estremamente ristretto o generare iterati instabili. Per questo motivo, il metodo di Newton è spesso combinato con altri metodi più robusti (come la bisezione) per ottenere un iniziale sufficientemente buono prima di applicare l'iterazione di Newton.

Velocità di convergenza. Sia α la radice esatta di $(f(x) = 0)$. Si può mostrare, espandendo $f(x)$ intorno a x_k con il resto di Lagrange e valutando in $x = \alpha$, che

$$\alpha - x_{k+1} = -\frac{1}{2}\frac{f''(\xi_k)}{f'(x_k)}(\alpha - x_k)^2, \quad \xi_k \in (x_k, \alpha),$$

da cui si ha

$$\lim_{k\to\infty} \frac{\alpha - x_{k+1}}{(\alpha - x_k)^2} = -\frac{1}{2}\frac{f''(\alpha)}{f'(\alpha)}$$

Dunque se la quantità $\frac{1}{2}\frac{f''(\alpha)}{f'(\alpha)}$ è uniformemente limitata, la convergenza del metodo di Newton è *quadratica*: una volta sufficientemente vicini alla soluzione, l'errore si annulla molto rapidamente.

Newton come iterata di punto fisso. Notiamo che il metodo di Newton può essere considerato come una iterata di punto fisso in cui si sceglie

$$g(x) = x - \frac{f(x)}{f'(x)}.$$

Avvertenze pratiche e criteri di arresto. Dal punto di vista computazionale, l'applicazione del metodo richiede la valutazione della derivata f'. Se $f'(x_k)$ è zero, o molto vicino allo zero, il passo di Newton diventa numericamente instabile: il denominatore piccolo può generare salti molto grandi o addirittura uscire dalla regione di convergenza. È dunque buona norma monitorare la dimensione di $f'(x_k)$ e predisporre un criterio di arresto di sicurezza quando $|f'(x_k)| < \varepsilon_d$, con ε_d opportunamente scelto.

Analogamente ad altri metodi iterativi, è necessario prevedere uno o più criteri di arresto. I più comuni per il metodo di Newton sono:

- Errore assoluto sull'iterato:
  $$|x_{k+1} - x_k| < \mathrm{Tol}_x,$$
  che controlla la stabilizzazione della successione degli iterati.
- Errore relativo sugli iterati:
  $$\frac{|x_{k+1} - x_k|}{|x_{k+1}|} < \mathrm{Tol}_{\mathrm{rel}},$$
  utile quando i valori coinvolti sono molto grandi o molto piccoli.
- Controllo del residuo:
  $$|f(x_{k+1})| < \mathrm{Tol}_f,$$
  che garantisce che il punto trovato sia effettivamente una radice approssimata.
- Criterio di sicurezza sulla derivata:
  $$|f'(x_k)| < \varepsilon_d,$$
  nel qual caso l'algoritmo deve fermarsi per evitare instabilità numerica.
- Numero massimo di iterazioni $k_{\max}$, per evitare loop infiniti in caso di mancata convergenza.

L'utilizzo congiunto di più criteri permette di aumentare l'affidabilità del metodo e prevenire situazioni di divergenza o stagnazione.

Algorithm 3 Metodo di Newton

Require: Funzione f, derivata f', stima iniziale x_0, tolleranze Tol_x, Tol_f, soglia derivata ε_d, massimo iterazioni $k_{\max}$

(Primo passo) Calcolare $f(x_0)$ e $f'(x_0)$
if $|f'(x_0)| < \varepsilon_d$ **then**
    return "Derivata troppo piccola: impossibile applicare Newton"
end if
for $k = 0, 1, 2, \ldots, k_{\max} - 1$ **do**
    Calcolare il nuovo iterato:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

    Calcolare $f(x_{k+1})$ e $f'(x_{k+1})$
    if $|f'(x_{k+1})| < \varepsilon_d$ **then**
        return "Derivata troppo piccola: possibile instabilità"
    end if
    (Criteri di arresto)
    if $\dfrac{|x_{k+1} - x_k|}{|x_{k+1}|} < \text{Tol}_x$ **then**
        return x_{k+1}
    end if
    if $|f(x_{k+1})| < \text{Tol}_f$ **then**
        return x_{k+1}
    end if
end for
return "Numero massimo di iterazioni raggiunto"

12.4 Osservazioni finali

Tutti gli algoritmi visti costruiscono successioni di stime iterativamente; non sempre una successione converge alla soluzione voluta. Il metodo meno schizzinoso è la bisezione (convergenza garantita se si ha un cambio di segno iniziale), mentre Newton è più veloce quando converge, ma richiede stime iniziali adeguate e/o informazioni sulla derivata. L'approccio di punto fisso è un utile strumento teorico e pratico quando si riesce a costruire una g contrattiva (ovvero tale che $|g'(x)| < 1$) nell'intorno della radice.

Laboratorio di Programmazione e Calcolo
Canale 2
Appunti del corso

Simone Cacace e Giuseppe Visconti
Dipartimento di Matematica
Sapienza Università di Roma

Anno Accademico 2025–2026

13 Interpolazione polinomiale

In questo capitolo studiamo il problema dell'interpolazione polinomiale che possiamo formulare in due modi.

1. Siano assegnati dei dati f_i, $i = 0 \dots n$ che rappresentano i valori di una funzione incognita f, definiti su un insieme di punti $x_0, \dots, x_n$. In questo caso, lo scopo è trovare un polinomio $P^n(x)$, di grado n, che interpola questi dati, nel senso che

$$P^n(x_i) = f(x_i), \quad i = 0, \dots n, \tag{2}$$

   e vogliamo usare P^n come approssimazione della funzione incognita f.

2. Oppure, conosciamo la funzione f, ma vogliamo approssimarla con un polinomio $P^n(x)$, perchè vogliamo utilizzare il polinomio per approssimare delle operazioni che su f sono complicate, mentre su P^n potrebbero essere molto più semplici. Per esempio, vogliamo approssimare l'integrale di f, che magari non siamo capaci di calcolare esattamente, ed utilizziamo P^n per approssimare la primitiva di f con $\int P^n$, che invece è molto facile da trovare.

Nel primo caso, conosciamo f solo sui nodi x_i (la griglia del problema di interpolazione), e siamo costretti ad utilizzare i nodi x_i anche se la griglia che abbiamo può presentare dei problemi. Nel secondo caso, conosciamo f e dunque possiamo calcolarla dove ci pare. In particolare, possiamo scegliere la griglia più conveniente per trovare un opportuno polinomio interpolante.

In questo capitolo, supporremo che è data una funzione $f(x)$, continua con tutte le sue derivate su un intervallo chiuso e limitato $[a, b] \in \mathbb{R}$. Dunque, studiamo soprattutto il secondo caso, e per questo ha senso chiedersi quale sarà la griglia più adatta per interpolare f. Se invece f non è nota, ma la conosciamo solo per punti, allora utilizzeremo gli stessi algoritmi, ma sapremo quali conseguenze aspettarci a seconda della distribuzione dei nodi x_i sui quali sono definiti i dati f_i.

13.1 I polinomi di base di Lagrange e il polinomio di interpolazione

Data una griglia $X = \{x_0 < x_1 < \cdots < x_n\}$, i polinomi di Lagrange sono una particolare base dello spazio $\mathbb{P}^n$, ovvero dei polinomi di grado $\leq n$, definita a partire dalla griglia di interpolazione. Chiamo polinomi di Lagrange, basati sulla griglia X, gli $n + 1$ polinomi ℓ_i definiti dalle condizioni

$$\ell_i(x_j) = \begin{cases} 0 & j \neq i \\ 1 & j = i. \end{cases} \qquad \forall\, x_j \in X. \tag{3}$$

I polinomi ℓ_i possono essere calcolati facilmente con la formula

$$\ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}. \tag{4}$$

Infatti, le condizioni di interpolazione (3) significano che ogni polinomio di Lagrange ha n radici reali e distinte, perché abbiamo supposto che i nodi della griglia siano distinti. Dunque, il polinomio ℓ_i può essere scritto in forma fattorizzata utilizzando tutti i nodi x_j sui quali vale zero, come

$$\ell_i(x) = C \prod_{j=0, j\neq i}^{n} (x - x_j).$$

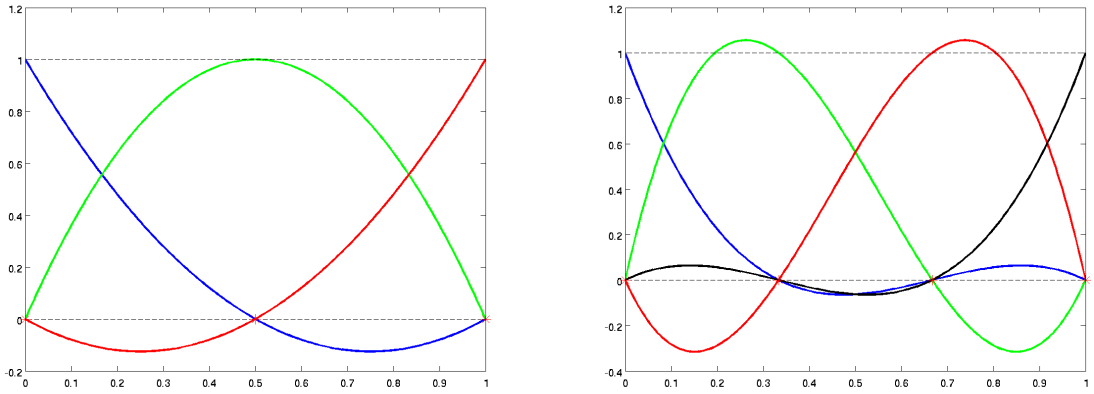


Figura 1: I 3 polinomi di Lagrange di grado 2 definiti su una griglia uniforme con 3 nodi (a sinistra) e i 4 polinomi di Lagrange di grado 3 definiti su una griglia uniforme con 4 nodi (a destra).

La costante C è univocamente determinata dall'unica condizione di interpolazione che non abbiamo ancora usato, cioè, sostituiamo nell'equazione qui sopra $x = x_i$. Otteniamo

$$1 = \ell_i(x_i) = C \prod_{j=0, j\neq i}^{n} (x_i - x_j),$$

da cui ricaviamo C. Abbiamo così dimostrato che vale l'equazione (4).

I polinomi di Lagrange formano una base di $\mathbb{P}^n$. E' facile verificare che ogni $\ell_i \in \mathbb{P}^n$. Dimostriamo che gli ℓ_i formano una base in $\mathbb{P}^n$. Sappiamo che $\dim(\mathbb{P}^n) = n+1$. Questo significa che ogni base di $\mathbb{P}^n$ è composta di $n+1$ elementi. Ora, i polinomi di Lagrange sono $n+1$, perchè su ognuno degli $n+1$ nodi della griglia X è definito un polinomio diverso. Dobbiamo però dimostrare che gli $\ell_i(x)$ sono indipendenti. Per far questo, dobbiamo dimostrare che se una loro combinazione lineare dà zero, allora tutti i coefficienti della combinazione lineare devono per forza essere nulli. Sia dunque

$$\sum_{i=0}^{n} c_i \ell_i(x) = 0 \quad \forall x \in [a, b].$$

Ma se questa espressione è zero ovunque, allora in particolare deve essere zero su qualunque nodo della griglia X. Consideriamo il nodo x_j, e sostituiamo nell'espressione qui sopra. Otteniamo

$$0 = \sum_{i=0}^{n} c_i \ell_i(x_j) = c_j \ell_j(x_j) = c_j,$$

dove ho usato il fatto che soltanto il polinomio $\ell_j(x)$ è diverso da zero sul nodo x_j e che $\ell_j(x_j) = 1$. Quindi, ogni $c_j = 0$, e dunque tutti i polinomi $\ell_i(x)$ sono linearmente indipendenti. Possiamo così concludere che i polinomi di Lagrange costituiscono una base per $\mathbb{P}^n$, ovvero ogni polinomio di $\mathbb{P}^n$ può essere scritto come combinazione lineare dei polinomi di Lagrange.

Proprietà dei polinomi di Lagrange. Per visualizzare i polinomi di Lagrange, devo calcolarli su una griglia molto più fitta di X. Infatti, se li disegnassi utilizzando solo i punti della griglia X

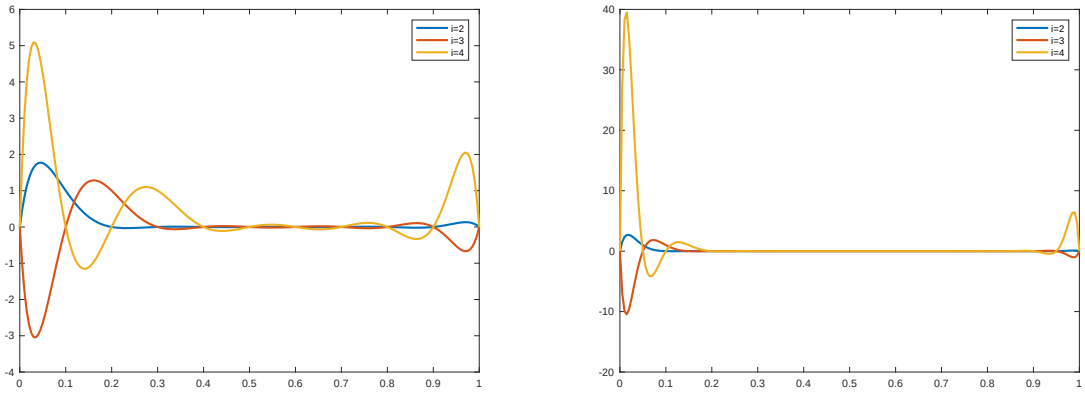


Figura 2: Polinomi di Lagrange di grado 10 definiti su una griglia uniforme con 11 nodi (a sinistra) e polinomi di Lagrange di grado 20 definiti su una griglia uniforme con 21 nodi (a destra). Sono rappresentati i polinomi ℓ_1, ℓ_2, ℓ_3, con la notazione usata in questa dispensa.

vedrei tutti zeri, ed un unico valore uguale a 1. La figura 1 mostra i polinomi di Lagrange che si ottengono con una griglia di 3 e di 4 nodi equispaziati. Nel primo caso, risultano 3 polinomi di grado 2, mentre nel secondo avremo 4 polinomi di grado 3. La figura successiva Fig. 2 mostra alcuni polinomi di Lagrange ottenuti su una griglia equispaziata con $n = 10$ e $n = 20$ nodi. Si può notare che i polinomi di Lagrange ora hanno forti oscillazioni, che crescono al crescere di n, vicino agli estremi dell'intervallo.

Esistenza del polinomio di interpolazione. Dal momento che i polinomi di Lagrange formano una base per $\mathbb{P}^n$, anche il polinomio di interpolazione di una funzione f sulla griglia X può essere scritto come combinazione lineare dei polinomi di Lagrange. E' facile verificare che il polinomio interpolatore può essere scritto esplicitamente con la formula

$$P^n(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x). \tag{5}$$

Basta verificare che valgono le condizioni di interpolazione, cioè che $p^n(x_j) = f(x_j), \forall j = 0, \ldots, n$. L'equazione (5) ha un'importante conseguenza: di fatto, questa equazione permette di scrivere esplicitamente il polinomio di interpolazione. Dunque questa equazione dimostra che il polinomio di interpolazione esiste purché i nodi della griglia X siano distinti.

Unicità del polinomio di interpolazione. Supponiamo che ci siano due polinomi P^n e Q^n, entrambi in $\mathbb{P}^n$ che soddisfano le condizioni di interpolazione (2). Calcoliamo la loro differenza $Z(x) = Q^n(x) - P^n(x)$. Poichè $\mathbb{P}^n$ è uno spazio lineare, anche $Z \in \mathbb{P}^n$. Ma, grazie alle condizioni di interpolazione, sappiamo che Z è nullo su tutti i nodi della griglia. Quindi, Z è un polinomio di grado al più n che ha $n+1$ radici distinte, e questo vuol dire che Z è identicamente nullo, o, equivalentemente, che P^n e Q^n sono identici.

Con questi ragionamenti, abbiamo dimostrato che esiste ed è unico il polinomio P^n di grado al più n che soddisfa le condizioni di interpolazione (2) su una griglia X composta di $n+1$ nodi distinti.

Costo computazionale. Supponiamo di dover calcolare il polinomio di interpolazione in diversi punti. Sia $M \gg 1$ il numero di punti nei quali deve essere calcolato il polinomio. Se

usiamo la formulazione di Lagrange, vediamo che per ogni punto richiesto, la valutazione di un singolo polinomio di Lagrange su un valore x richiede $O(n)$ operazioni. Per valutare la (5) è necessario calcolare $n+1$ polinomi. Dunque il costo per un singolo punto è $O(n^2)$, e il costo computazionale per il calcolo del polinomio sugli M punti richiesti diventa $O(n^2M)$.

Quindi, se il polinomio deve essere calcolato in pochi punti, il metodo di Lagrange non è computazionalmente costoso. Invece, se $M \gg n$, come accade spesso, l'interpolazione di Lagrange risulta avere un alto costo computazionale.

Confrontare il polinomio di interpolazione con la funzione interpolata. Per studiare l'andamento dell'interpolazione polinomiale, dobbiamo confrontare il polinomio di interpolazione P^n con la funzione f dalla quale abbiamo ricavato i dati. Sulla griglia di interpolazione, polinomio e funzione coincidono per costruzione. Quindi, dobbiamo definire sempre sull'intervallo $[a,b]$ una griglia Y molto più fitta di quella che abbiamo usato per costruire il polinomio di interpolazione, per poter vedere se il polinomio approssima bene la funzione anche per valori di y diversi da quelli dei nodi di interpolazione.

Valutiamo poi sia il polinomio che la funzione sulla griglia fitta e disegniamo il grafico delle due funzioni. In modo più quantitativo, possiamo definire l'errore fra il polinomio e la funzione stimando la norma infinito dell'errore, calcolando la grandezza

$$e_n = \max_{y \in Y} |f(y) - P^n(y)|.$$

13.2 Griglie di interpolazione

I polinomi di Lagrange dipendono moltissimo dal tipo di griglia su cui sono definiti. Abbiamo visto nella figura 2 che i polinomi di Lagrange hanno forti oscillazioni in prossimità degli estremi dell'intervallo su cui sono definiti, se utilizzo una griglia di nodi uniformemente distribuiti su $[a,b]$.

Si può verificare facilmente che i polinomi di Lagrange si mantengono limitati se utilizzo una griglia caratterizzata da avere nodi molto più fitti in prossimità degli estremi dell'intervallo. In particolare, consideriamo la griglia dei nodi di Chebyshev definiti da

$$x_j = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{2j+1}{2(n+1)}\pi\right), \qquad j = 0, \ldots, n. \tag{6}$$

La griglia di Chebyshev si ottiene dalle radici di insiemi di polinomi ortogonali, i polinomi di Chebyshev. La teoria dei polinomi ortogonali è molto importante in teoria dell'approssimazione di funzioni, ma va decisamente al di là degli argomenti trattati in un corso introduttivo come questo.

13.3 Errore nell'interpolazione polinomiale

Il polinomio P^n di interpolazione di una funzione f spesso è calcolato per avere un'approssimazione della funzione f, anche quando f è nota. Ma quanto è accurata questa approssimazione?

Sappiamo già che il polinomio P^n che interpola una funzione f su una griglia di $n+1$ nodi distinti esiste, è unico e soddisfa il sistema di equazioni

$$P^n(x_i) = f(x_i), \quad \forall i = 0, \ldots, n.$$

Quindi, l'errore può essere scritto come

$$e_n(x) = f(x) - P^n(x) = \prod_{i=0}^{n}(x - x_i)\, R(x), \tag{7}$$

dove sono state enfatizzate le $n+1$ radici della funzione $e_n(x)$. La funzione che raccoglie il contributo di queste radici si chiama *funzione nodale* ed è completamente definita dalla griglia di interpolazione tramite la formula

$$\omega_{n+1}(x) = \prod_{i=0}^{n}(x - x_i). \tag{8}$$

La formula (7) dà zero per ogni $x = x_i$ appartenente alla griglia di interpolazione, grazie alla funzione nodale.

Teorema 3 (Errore di interpolazione)**.** *Sia $P^n(x)$ il polinomio di grado al più n che interpola la funzione f sui nodi distinti $\mathbf{x} = [x_0, \ldots, x_n] \in [a, b]$. Supponiamo che f abbia almeno $n+1$ derivate continue. L'errore di interpolazione è dato da*

$$e_n(x) = f(x) - P^n(x) = \frac{1}{(n+1)!}\prod_{i=0}^{n}(x - x_i) f^{(n+1)}(\xi), \quad \forall x \in [a, b], \tag{9}$$

per qualche $\xi = \xi(x) \in [a, b]$.

Dimostrazione. Supponiamo di fissare un valore $x \neq x_i, i = 0, \ldots, n$ e cerchiamo di calcolare $R(x)$ in (7). Definiamo la funzione ausiliaria

$$G(t) = e_n(t) - \omega_n(t)R(x).$$

Si vede facilmente che $G(t)$ ha $n+2$ zeri distinti. Infatti $G(x_i) = 0$ per $i = 0, \ldots, n$, e abbiamo così $n+1$ zeri. Inoltre, per costruzione, quando $t = x$, otteniamo $G(x) = 0$. Per il teorema di Rolle, $G'(t)$ avrà $n+1$ zeri, e, ripetendo questo ragionamento, troviamo che $G^{(n+1)}$ ha uno zero in (a, b). Sia dunque ξ l'unico zero di $G^{(n+1)}$ e calcoliamo

$$G^{(n+1)}(\xi) = 0 = f^{(n+1)}(\xi) - (n+1)!R(x)$$

che ci permette di calcolare $R(x)$. □

E' importante notare che l'equazione (9) per l'errore nell'interpolazione polinomiale *non è* un risultato di convergenza. Innanzitutto, non è detto che le derivate di ordine alto di una funzione siano limitate. Ci aspettiamo dunque che l'interpolazione polinomiale fornirà pessimi risultati su funzioni non regolari. Vediamo alcuni risultati nella figura 3.

Ma anche se la funzione avesse tutte le derivate limitate, non è detto che l'errore converga a zero per $n \to \infty$. Il comportamento dell'errore dipende infatti anche dalla funzione nodale. E si vede facilmente che la funzione nodale ha un comportamento molto diverso a seconda della griglia di interpolazione.

La fig. 4 mostra l'andamento della funzione nodale sull'intervallo $[-2, 2]$, con 10 e con 20 nodi distribuiti uniformemente sull'intervallo. Si vede chiaramente che la funzione nodale cresce velocemente aumentando n e che i picchi più grandi sono localizzati agli estremi dell'intervallo. Invece, se i nodi sono distribuiti come in una griglia di Chebyshev, la funzione nodale si mantiene limitata, come si vede nella fig. 5.

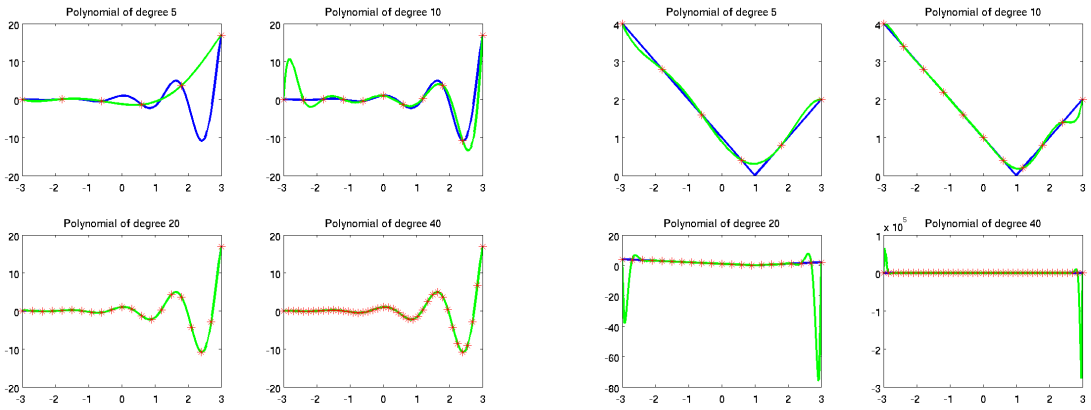


Figura 3: Interpolazione di funzioni con polinomi di grado elevato. A sinistra, la funzione è del tipo $e^x \sin(kx)$ e ha dunque infinite derivate limitate sull'intervallo $[-3, 3]$ sul quale è stata interpolata la funzione. A destra, interpolazione della funzione non regolare $f(x) = |x - 1|$. La funzione è disegnata in blu, mentre il polinomio è in verde. Gli asterischi rossi sono i punti di interpolazione. La griglia è uniforme su $[-3, 3]$.

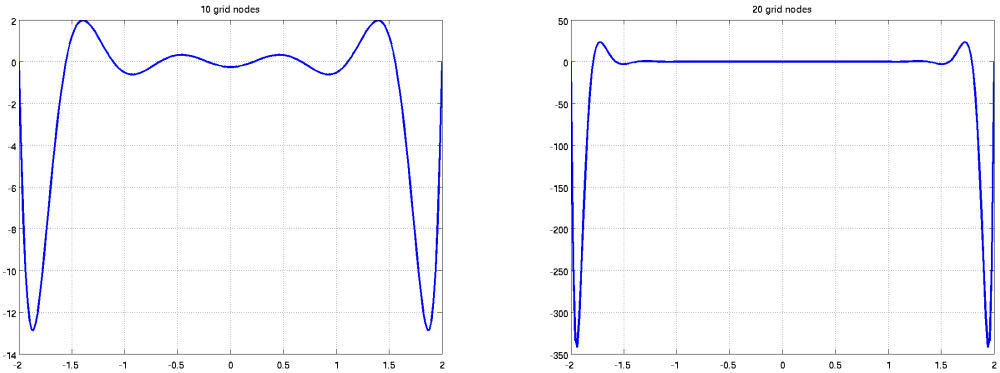


Figura 4: Funzione nodale su una griglia uniforme. A sinistra, 10 nodi, a destra 20 nodi.

13.4 Condizionamento nell'interpolazione polinomiale

Supponiamo di interpolare dei dati $(x_i, f(x_i))$, $i = 0, \dots n$ ottenendo il polinomio di grado n $P^n(x)$. Supponiamo ora che i dati siano inquinati da un errore δ_i, $i = 0, \dots n$. Ci chiediamo quanto il polinomio di interpolazione sia sensibile all'errore sui dati. Cioè ci chiediamo qual è il *condizionamento* per l'interpolazione polinomiale. Per formalizzare meglio il problema sia dunque

$$\begin{aligned} P^n(x) &\quad \text{il polinomio che interpola i dati } (x_i, f(x_i)),\ i = 0, \dots n \\ \tilde{P}^n(x) &\quad \text{il polinomio che interpola i dati } (x_i, \tilde{f}(x_i)),\ i = 0, \dots n, \end{aligned} \tag{10}$$

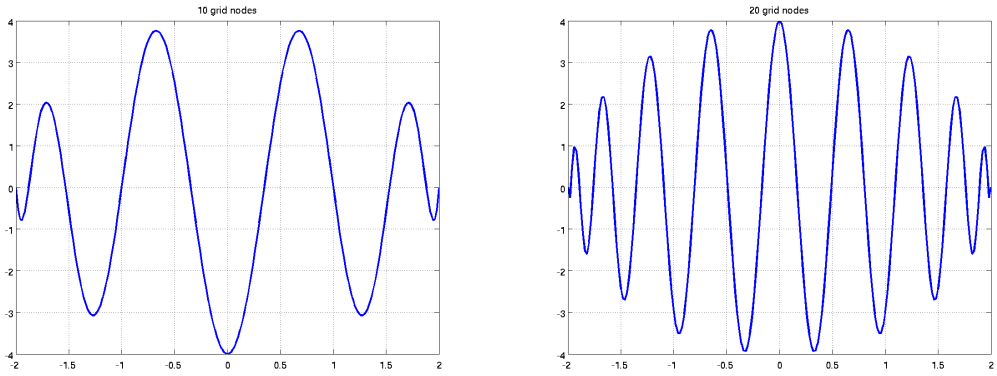


Figura 5: Funzione nodale su una griglia di Gauss Lobatto. A sinistra, 10 nodi, a destra 20 nodi.

dove $\tilde{f}(x_i) = f(x_i) + \delta_i$ sono i dati perturbati. Vogliamo valutare quanto è grande $||P^n(x) - \tilde{P}^n(x)||$. Calcoliamo dunque

$$\begin{aligned} |P^n(x) - \tilde{P}^n(x)| &= \left| \sum_{i=0}^{n} f(x_i)\ell_i(x) - \sum_{i=0}^{n} \tilde{f}(x_i)\ell_i(x) \right| \\ &= \left| \sum_{i=0}^{n} \left[f(x_i) - \tilde{f}(x_i) \right] \ell_i(x) \right| \\ &\leq \sum_{i=0}^{n} \left| f(x_i) - \tilde{f}(x_i) \right| |\ell_i(x)| \\ &\leq \max_{0 \leq i \leq n} \left| f(x_i) - \tilde{f}(x_i) \right| \sum_{i=0}^{n} |\ell_i(x)| \,. \end{aligned}$$

Il vettore di componenti $f(x_i) - \tilde{f}(x_i) = \delta_i$ è l'errore nei dati. La sua norma infinito $||\delta||_\infty = \max_i |\delta_i|$ è la prima parte della formula sopra. Dunque

$$|P^n(x) - \tilde{P}^n(x)| \leq ||\delta|| \sum_{i=0}^{n} |\ell_i(x)| \leq \Lambda_n(\mathbf{X}) ||\delta||, \tag{11}$$

dove la grandezza

$$\Lambda_n(\mathbf{X}) = \left|\left| \sum_{i=0}^{n} |\ell_i(x)| \right|\right|_\infty = \max_{x_0 \leq x \leq x_n} \sum_{i=0}^{n} |\ell_i(x)| \tag{12}$$

è chiamata costante di Lebesgue, calcolata sulla griglia $\mathbf{X}$ di $n+1$ nodi e gli ℓ_i sono i polinomi di Lagrange definiti sulla griglia di interpolazione $\mathbf{X}$. Notare che la costante di Lebesgue dipende solamente da come sono scelti i nodi della griglia $\mathbf{X}$.

Quindi,

$$||P^n - \tilde{P}^n||_\infty = \max_{x \in [x_0 . x_n]} |p^n(x) - \tilde{p}^n(x)| \leq \Lambda_n(\mathbf{X}) ||\delta||. \tag{13}$$

In altre parole, l'errore nei dati di interpolazione si trasferisce nell'errore sul polinomio di interpolazione con un fattore di amplificazione che è dato dalla costante di Lebesgue.

La costante di Lebesgue cresce molto velocemente con n su una griglia uniforme, mentre cresce molto lentamente su una griglia di tipo Chebyshev (vedi esercizio E1 della scheda di laboratorio).

Laboratorio di Programmazione e Calcolo
Canale 2
Appunti del corso

Simone Cacace e Giuseppe Visconti
Dipartimento di Matematica
Sapienza Università di Roma

Anno Accademico 2025–2026

14 Integrazione numerica

Il problema del calcolo di un integrale definito

$$I = \int_a^b f(x)\,\mathrm{d}x,$$

dove f è un'arbitraria funzione continua in $[a, b]$, è centrale in analisi e in numerose applicazioni scientifiche. Nonostante l'apparente semplicità formale, nella pratica è spesso impossibile determinare in modo esplicito la primitiva di una funzione. Anche quando una primitiva è nota, la sua valutazione può risultare onerosa o numericamente instabile.

Un esempio emblematico è il seguente:

$$\frac{1}{\pi}\int_0^{\pi} \cos(4x)\,\cos(3\sin x)\,dx = \left(\frac{3}{2}\right)^4 \sum_{k=0}^{\infty} \frac{(-9/4)^k}{k!\,(k+4)!},$$

dove il problema dell'integrazione viene trasformato nel problema della valutazione di una serie potenzialmente difficile da sommare con precisione. Questo mette in luce come, anche quando una formula analitica esiste, essa non rappresenti necessariamente il metodo migliore per ottenere una buona approssimazione numerica.

Nella maggior parte delle applicazioni reali, il problema è ancora più impegnativo: la funzione da integrare può non essere nota in forma analitica, ma solo tramite un insieme finito di dati sperimentali o numerici,

$$(x_0, f_0),\ (x_1, f_1), \ldots, (x_n, f_n).$$

In tali contesti non solo è impossibile ottenere una primitiva, ma non è nemmeno definita la funzione tra i punti di misura. Si rende quindi necessario l'uso di metodi numerici che:

- non richiedano la conoscenza della primitiva;
- siano robusti rispetto al rumore nei dati;
- forniscano una stima dell'errore;
- permettano di controllare la precisione tramite il raffinamento della discretizzazione.

L'obiettivo generale è quello di calcolare un'approssimazione dell'integrale nella forma

$$I \approx Q(f),$$

dove $Q(f)$ è una *formula di quadratura*, ottenuta sostituendo l'integrale con una combinazione di valori di f calcolati in un insieme finito di punti:

$$Q(f) = \sum_{k=0}^{m} w_k\, f(x_k).$$

Le formule di quadratura possono essere:

- *a un punto* (regola del punto medio),
- *a due punti* (regola dei trapezi),
- *a più punti* (famiglia delle formule di Newton–Cotes),
- *stocastiche* (quadratura Monte Carlo).

Nel capitolo che segue introdurremo tali metodi e ne analizzeremo l'errore.

14.1 La formula del punto medio

Una semplice procedura per approssimare I consiste nel suddividere l'intervallo $[a, b]$ in sottointervalli $\Omega_k = [x_{k-1}, x_k]$, $k = 1, \ldots, M$, con $x_k = a + kh$, $k = 0, \ldots, M$, $h = (b-a)/M$ ampiezza di ogni sottointervallo.

Grazie all'additività degli integrali possiamo scrivere

$$I = \int_a^b f(x)\,\mathrm{d}x = \sum_{k=1}^{M} \int_{\Omega_k} f(x)\,\mathrm{d}x.$$

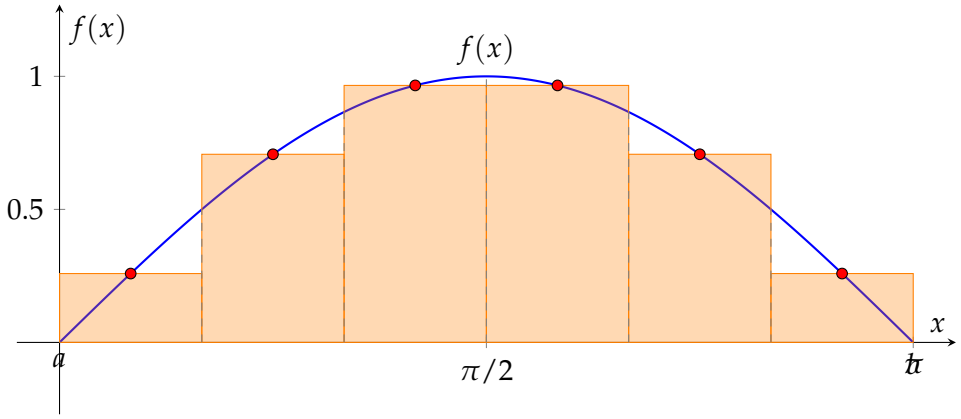
A questo punto, su ogni intervallo Ω_k si sostituisce l'integrale di f con l'integrale di un polinomio p che approssimi f su Ω_k. La soluzione più semplice consiste nell'impiegare il polinomio costante che interpola f nel punto medio dell'intervallo Ω_k:

$$c_k = \frac{x_{k-1} + x_k}{2}.$$

In questo modo si ottiene la formula (composita) del punto medio

$$I \approx Q(f) = h \sum_{k=1}^{M} f(c_k).$$

Tale formula è nota anche come *formula dei rettangoli* per la sua interpretazione geometrica.



Ha ordine di accuratezza 2 rispetto ad h, ovvero errore $O(h^2)$. Precisamente, se f è derivabile con continuità fino al second'ordine, si ha

$$I - Q(f) = \frac{b-a}{24} h^2 f''(\xi), \qquad \xi \in (a, b).$$

Osservazione. La scelta del punto medio rispetto ai valori ai capi dell'intervallo migliora l'accuratezza della formula. Per comprenderne il motivo, si consideri lo sviluppo di Taylor di f attorno al punto medio c_k di ciascun sottointervallo Ω_k:

$$f(x) = f(c_k) + f'(c_k)(x - c_k) + \frac{f''(\xi_x)}{2}(x - c_k)^2,$$

dove $\xi_x \in \Omega_k$. Integrando membro a membro su $\Omega_k = [c_k - h/2, c_k + h/2]$ si ottiene

$$\int_{\Omega_k} f(x)\,\mathrm{d}x = hf(c_k) + f'(c_k) \int_{\Omega_k} (x - c_k)\,\mathrm{d}x + \frac{1}{2} \int_{\Omega_k} f''(\xi_x)(x - c_k)^2\,\mathrm{d}x.$$

Il secondo termine è nullo perché l'integrale di una funzione dispari su un intervallo simmetrico rispetto all'origine vale zero:

$$\int_{c_k-h/2}^{c_k+h/2} (x - c_k)\,\mathrm{d}x = 0.$$

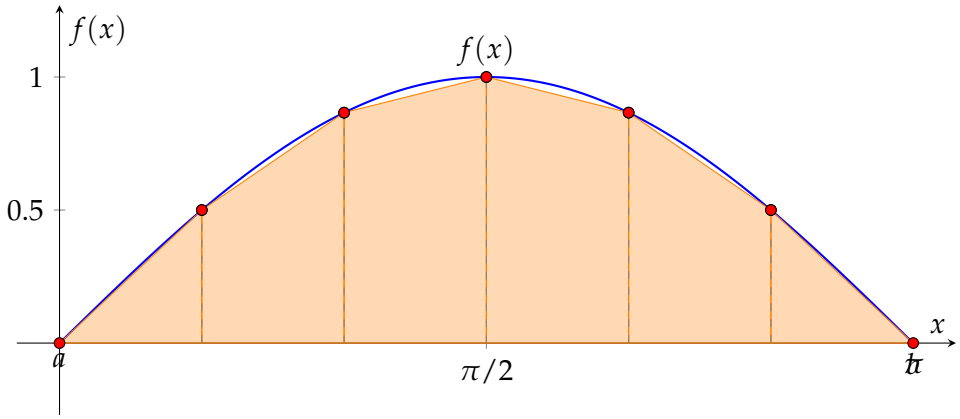
Il primo termine coincide con l'approssimazione di quadratura, mentre l'ultimo termine è proporzionale a h^3. Ne segue che l'errore locale è $O(h^3)$ e, sommando su tutti i $M \approx (b-a)/h$ sottointervalli, si ottiene un errore globale $O(h^2)$.

Questo annullamento del termine lineare dello sviluppo di Taylor spiega perché la formula del punto medio ha ordine 2, mentre le formule dei rettangoli "a sinistra" o "a destra" possiedono solo ordine 1.

14.2 La formula del trapezio

Qualora per approssimare l'integrale di f su Ω_k si usi il polinomio interpolatore di grado 1 nei nodi estremi x_{k-1} e x_k, si perviene alla cosidetta *formula del trapezio*:

$$Q(f) = \frac{h}{2}\sum_{k=1}^{M}\left(f(x_k) + f(x_{k-1})\right) = \frac{h}{2}\left(f(a) + f(b)\right) + h\sum_{k=1}^{M-1} f(x_k).$$



La formula del trapezio ha ordine di accuratezza 2. Infatti, si può dimostrare che per funzioni di classe $C^2(a,b)$ l'errore vale

$$I - Q(f) = -\frac{b-a}{12}h^2 f''(\xi), \qquad \xi \in (a,b).$$

Una semplice modifica di tale formula fornisce risultati più soddisfacenti. Sostituendo su ogni intervallo Ω_k la funzione f con il suo polinomio interpolatore di grado 1, calcolato però nei cosidetti *nodi di Gauss*:

$$\gamma_{k,1} = x_{k-1} + \left(1 - \frac{1}{\sqrt{3}}\right)\frac{h}{2}$$
$$\gamma_{k,2} = x_{k-1} + \left(1 + \frac{1}{\sqrt{3}}\right)\frac{h}{2}$$

si ottiene la formula

$$Q(f) = \frac{h}{2}\sum_{k=1}^{M}\left(f(\gamma_{k,1}) + f(\gamma_{k,2})\right).$$

Questa formula, detta *di Gauss*, ha ordine di accuratezza 4, in quanto:

$$I - Q(f) = \frac{b-a}{69120}h^4 f^{(4)}(\xi), \qquad \xi \in (a,b).$$

14.3 La formula di Simpson

Procediamo in modo del tutto analogo a quanto fatto per ricavare la formula del trapezio. L'unica differenza è che ora, su ogni intervallo Ω_k, la funzione f verrà sostituita con il polinomio interpolatore di grado 2 nei 3 nodi x_{k-1}, $c_k = (x_{k-1} + x_k)/2$ e x_k. Si ottiene allora la *formula di Simpson*:

$$Q(f) = \frac{h}{6} \sum_{k=1}^{M} \left(f(x_{k-1}) + 4f(c_k) + f(x_k) \right).$$

In questo caso, abbiamo ordine di accuratezza 4. Per $f \in C^4(a, b)$ vale infatti

$$I - Q(f) = -\frac{b-a}{180} h^4 f^{(4)}(\xi), \qquad \xi \in (a, b).$$

14.4 Quadratura Monte Carlo

L'idea della quadratura Monte Carlo è molto diversa dalle formule deterministiche viste in precedenza (punto medio, trapezio, Simpson). Qui non costruiamo polinomi o rettangoli che seguono la forma di f. Invece, immaginiamo di racchiudere il grafico della funzione dentro ad un grande rettangolo e di "sparare" punti a caso al suo interno. Contando quanti punti cadono sotto la curva, otteniamo una stima dell'area.

Consideriamo un intervallo $[a, b]$ e una funzione continua e non negativa $f(x)$ su tale intervallo. Per prima cosa costruiamo un rettangolo che contenga tutto il grafico, ad esempio

$$R = [a, b] \times [0, f_{\max}], \qquad f_{\max} = \max_{x \in [a,b]} f(x).$$

L'area di questo rettangolo è semplicemente $A_R = (b - a) f_{\max}$.

A questo punto, scegliamo un numero N di punti distribuiti "a caso" dentro il rettangolo R. Per ogni punto verifichiamo se si trova *sotto* il grafico della funzione, cioè se la sua coordinata verticale y soddisfa

$$y \leq f(x).$$

Indichiamo con K il numero di punti che cadono sotto la curva. Allora la frazione

$$\frac{K}{N}$$

rappresenta "quanto" del rettangolo è occupato dalla funzione, ovvero

$$\frac{I}{A_R} \approx \frac{K}{N}.$$

Per ottenere una stima dell'integrale calcoliamo quindi

$$Q(f) = \frac{K}{N} A_R.$$

Questa procedura prende il nome di *quadratura Monte Carlo*. È estremamente semplice da programmare e funziona bene anche in dimensioni alte (dove gli altri metodi diventano complicati), pur essendo meno precisa dei metodi deterministici in una dimensione.

Osservazione. Aumentando il numero di punti N, la stima tende a migliorare perché il campionamento diventa più uniforme all'interno del rettangolo. In pratica, più "spariamo" punti, più l'immagine media che otteniamo dell'area diventa accurata.

Laboratorio di Programmazione e Calcolo
Canale 2
Appunti del corso

Simone Cacace e Giuseppe Visconti
Dipartimento di Matematica
Sapienza Università di Roma

Anno Accademico 2025–2026

15 Sistemi lineari

La risoluzione di sistemi lineari

$$Ax = b, \qquad A \in \mathbb{R}^{n \times n},\ x, b \in \mathbb{R}^n$$

è uno dei problemi centrali dell'intera matematica computazionale. Sistemi di questo tipo compaiono in fisica, ingegneria, economia, grafica computerizzata, statistica e in qualunque modello che descriva fenomeni tramite equazioni. Nelle applicazioni reali la matrice A può essere molto grande, oppure può avere una struttura particolare (ad esempio essere sparsa, triangolare o simmetrica), oppure i dati possono essere affetti da errori di misura. Per questi motivi non è sempre possibile – o conveniente – risolvere il sistema con le sole tecniche dell'algebra lineare studiate nei corsi di base.

In questo capitolo introdurremo gli strumenti numerici fondamentali per affrontare il problema in modo automatico e stabile. Studieremo due principali famiglie di metodi:

- **Metodi iterativi**, come Jacobi e Gauss–Seidel, che costruiscono una sequenza di approssimazioni sempre migliori della soluzione.
- **Metodi diretti**, come l'eliminazione di Gauss o la fattorizzazione LU, che trasformano il sistema in uno equivalente e lo risolvono in un numero finito di operazioni.

Nel corso della trattazione vedremo come valutare l'accuratezza della soluzione numerica, come stimare l'errore e quando un metodo converge. Analizzeremo inoltre quali proprietà della matrice influenzano il comportamento dei metodi.

15.1 Risoluzione di sistemi lineari triangolari

Cominciamo a descrivere alcuni algoritmi per risolvere sistemi lineari algebrici del tipo $Ax = b$.

I casi più semplici sono quelli in cui A è una matrice triangolare. Diciamo che A è *triangolare inferiore* se $a_{i,j} = 0\ \forall j > i$, mentre A è *triangolare superiore* se $a_{i,j} = 0\ \forall i > j$. In entrambi questi casi, il sistema si risolve usando il metodo delle sostituzioni successive, come nei due algoritmi che seguono.

Algorithm 4 Risoluzione di un sistema triangolare inferiore

Require: A matrice $n \times n$, b vettore $n \times 1$
Ensure: x vettore soluzione, tale che $Ax = b$

```
x_1 = b_1 / a_{1,1}
for i = 2 ... n do
    s = 0
    for j = 1 ... i − 1 do
        s = s + a_{i,j} x_j
    end for
    x_i = (b_i − s) / a_{i,i}
end for
```

Algorithm 5 Risoluzione di un sistema triangolare superiore

Require: A matrice $n \times n$, b vettore $n \times 1$
Ensure: x vettore soluzione, tale che $Ax = b$
1: $x_n = b_n / a_{n,n}$
2: **for** $i = n - 1 \dots 1$ **with step** -1 **do**
3: $\quad s = 0$
4: $\quad$ **for** $j = i + 1 \dots n$ **do**
5: $\quad\quad s = s + a_{i,j} x_j$
6: $\quad$ **end for**
7: $\quad x_i = (b_i - s) / a_{i,i}$
8: **end for**

Una matrice triangolare è invertibile, e dunque il sistema ha un'unica soluzione, se $a_{i,i} \neq 0 \ \forall i$. Questa proprietà si ritrova negli algoritmi per la risoluzione dei sistemi triangolari, perché è necessario calcolare divisioni nelle quali ognuno degli elementi diagonali prima o poi si trova a denominatore.

15.2 Metodi iterativi

Come detto, nei metodi iterativi, si cerca di costruire una successione $\{x^{(k)}\} \in \mathbb{R}^n$, concepita in modo tale da avere $x^{(k)} \to x$ (in una distanza opportuna), dove x è la soluzione esatta, per $k \to \infty$. Come in tutti i metodi iterativi, definiremo dei criteri di stop per fermare l'iterazione quando l'errore, o il residuo, saranno abbastanza piccoli.

Strategia generale. Per costruire tale successione consideriamo una decomposizione della matrice

$$A = M - N,$$

dove M è una matrice non singolare scelta in modo da essere facilmente invertibile (ad esempio triangolare o addirittura diagonale), mentre N contiene il resto della struttura di A.

Riscrivendo il sistema si ottiene

$$Mx = Nx + b,$$

e moltiplicando per M^{-1} si ricava una forma equivalente

$$x = M^{-1}Nx + M^{-1}b.$$

A questo punto si introduce una successione di vettori $\{x^{(k)}\}_{k \geq 0}$ definita ricorsivamente da

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b, \tag{14}$$

partendo da una scelta iniziale arbitraria $x^{(0)}$.

I metodi iterativi più noti, come Jacobi e Gauss–Seidel, si ottengono scegliendo in modo diverso la decomposizione $A = M - N$; tali scelte determinano la forma esplicita del procedimento iterativo.

15.2.1 Metodo di Jacobi

Nel metodo di Jacobi, si sceglie M come la matrice diagonale che contiene la diagonale principale di A, e N contiene tutto il resto della matrice A, $N = M - A$. Poiché M è diagonale, la

risoluzione del sistema (14) è immediata. Componente per componente ottengo

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k \right).$$

Naturalmente, tutti gli elementi sulla diagonale di A devono essere non nulli, altrimenti si devono operare degli scambi di riga.

15.2.2 Metodo di Gauss-Seidel

Nel metodo di Gauss-Seidel si utilizza una scelta diversa per la decomposizione di A. Si scrive

$$A = M - N,$$

dove M è la matrice triangolare inferiore di A, comprensiva della diagonale, mentre N contiene gli elementi sopra la diagonale. Poiché M è triangolare inferiore, risolvere questo sistema è semplice: le componenti di $x^{(k+1)}$ vengono calcolate una alla volta, partendo da $i = 1$ e procedendo verso il basso, usando immediatamente i valori già aggiornati.

La formula esplicita per la i-esima componente diventa

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right).$$

Rispetto a Jacobi, qui i valori più recenti sono incorporati subito nel calcolo, il che in pratica rende il metodo spesso più veloce nel "correggere" gli errori. Anche in questo caso tutti gli elementi diagonali a_{ii} devono essere non nulli; in caso contrario è necessario applicare scambi di riga per ripristinare questa condizione.

15.2.3 Osservazioni intuitive sulla convergenza dei metodi iterativi

Quando si applica un metodo iterativo, ad esempio il metodo di Jacobi o di Gauss–Seidel, non si ottiene subito la soluzione del sistema lineare, ma si costruisce una successione di vettori

$$x^{(0)},\ x^{(1)},\ x^{(2)},\ \ldots$$

che, se il metodo funziona correttamente, si avvicinano progressivamente alla soluzione esatta.

Per studiare la convergenza del metodo iterativo si introduce il *vettore errore* al passo k:

$$e^{(k)} = x^{(k)} - x,$$

dove x è la soluzione esatta del sistema lineare. L'idea è che, se il metodo è efficace, i vettori $e^{(k)}$ devono tendere al vettore nullo.

In questo corso non disponiamo ancora degli strumenti matematici necessari per un'analisi completa della convergenza (come norme vettoriali, autovalori e raggio spettrale). Presentiamo quindi una descrizione qualitativa, utile per comprendere come e perché l'iterazione possa avvicinarsi alla soluzione.

Differenze tra iterazioni successive. Un primo modo per valutare se l'iterazione sta "funzionando" consiste nell'osservare la variazione tra due iterazioni consecutive. Se i vettori

$$x^{(k+1)} \quad \text{e} \quad x^{(k)}$$

diventano sempre più simili, significa che l'algoritmo sta stabilizzando la propria stima della soluzione. In pratica si controlla che la quantità

$$\max_{i=1,\dots,n} |x_i^{(k+1)} - x_i^{(k)}|$$

diventa piccola. Questo criterio è semplice, ma non garantisce sempre da solo che ci si stia avvicinando alla soluzione esatta.

Il residuo come indicatore affidabile. Un altro modo, più informativo, consiste nel calcolare il *residuo*

$$r^{(k)} = b - Ax^{(k)}.$$

Se $x^{(k)}$ fosse la soluzione esatta, il residuo sarebbe identicamente nullo. Durante l'iterazione, se il metodo è efficace, il residuo diventa sempre più piccolo. Nella pratica computazionale si usa proprio il residuo per stabilire un criterio di arresto, ad esempio imponendo che

$$\max_{i=1,\dots,n} |r_i^{(k)}| < \varepsilon,$$

con ε tolleranza fissata.

Perché a volte la convergenza c'è e a volte no. Non tutti i sistemi lineari consentono una convergenza rapida dei metodi iterativi. Tuttavia, alcune proprietà strutturali della matrice A favoriscono il buon comportamento dell'algoritmo. Un caso particolarmente importante è quello delle matrici *a diagonale dominante*, cioè matrici per le quali, in ogni equazione, il coefficiente della variabile principale è "più grande" della somma dei coefficienti delle altre variabili. In queste situazioni l'iterazione tende a essere stabile e a correggere progressivamente l'errore iniziale.

Sperimenteremo in laboratorio che una matrice con forte diagonale dominante conduce tipicamente a iterazioni che convergono rapidamente, mentre una matrice con struttura più "sbilanciata" può dar luogo a convergenza lenta o addirittura al fallimento del metodo.

Limiti della nostra trattazione. Una teoria completa della convergenza richiede strumenti più avanzati (norme, analisi spettrale), che saranno affrontati in corsi successivi. In queste dispense adotteremo dunque un punto di vista operativo: valuteremo la convergenza osservando il comportamento del residuo e confronteremo sperimentalmente l'efficacia dei vari metodi in funzione della struttura della matrice.

In sintesi, per i nostri scopi è sufficiente ricordare che:

- un metodo iterativo converge quando le approssimazioni successive si stabilizzano;
- il residuo $r^{(k)} = b - Ax^{(k)}$ è l'indicatore più utile per controllare l'avvicinamento alla soluzione;
- matrici a diagonale dominante favoriscono fortemente la convergenza.

La condizione sulla differenza tra iterate successive e la condizione sul residuo, unite alla richiesta di un numero massimo di iterazioni, forniscono criteri di arresto per i metodi iterativi.

15.3 Metodi diretti: algoritmo di eliminazione di Gauss

Un modo per risolvere il sistema lineare $Ax = b$ è di utilizzare l'algoritmo di eliminazione di Gauss che applica delle trasformazioni lineari ad entrambi i membri del sistema, in modo da ottenere il sistema equivalente

$$Ux = \tilde{b},$$

dove U è una matrice triangolare superiore e $\tilde{b}$ è il vettore che si ottiene applicando a b tutte le trasformazioni che applichiamo ad A. Dunque, se

$$MAx = Mb, \quad M \text{ tale che } MA = U \quad \Longrightarrow \quad Ux = Mb.$$

allora $\tilde{b} = Mb$. La matrice M raggruppa tutte le trasformazioni che sono state applicate ad A per trasformarla nella matrice U.

Supponiamo che la matrice A abbia come elementi a_{ij}, $i, j = 1, \ldots, n$. Denotiamo A_i la riga i della matrice A, mentre $A^{(k)}$ denoterà la matrice che si ottine da A all'k-esimo stadio della trasformazione. Dunque, $A^{(0)} = A$.

Nell'algoritmo di Gauss, si procede colonna per colonna. Si comincia inserendo degli zeri nella prima colonna di A, sotto l'elemento diagonale. Per ottenere questo risultato con una trasformazione lineare, innanzitutto si calcola il *moltiplicatore* $m_{21} = a_{21}/a_{11}$.

Si noti che a_{11} deve essere diverso da zero. Se così non fosse, l'algoritmo non potrebbe procedere, poiché il moltiplicatore m_{21} non sarebbe definito. In questo caso si applica una *permutazione di righe*: si scambia la prima riga con una riga successiva $i > 1$ tale che $a_{i1} \neq 0$. Questa operazione prende il nome di *pivoting*.

In pratica, prima di iniziare l'eliminazione sulla colonna 1, si cerca un elemento non nullo (o sufficientemente grande in valore assoluto) nella stessa colonna e lo si porta in posizione $(1, 1)$ mediante uno scambio di righe.

Da ora in poi, supponiamo che $a_{11} \neq 0$.

Moltiplichiamo la riga $A_1^{(0)}$ per m_{21} e sottraggo dalla riga $A_2^{(0)}$. In questo modo otteniamo

$$A_2^{(1)} = A_2^{(0)} - m_{21} A_1^{(0)} = \left[0, a_{22}^{(1)}, \cdots, a_{2n}^{(1)},\right], \qquad a_{2j}^{(1)} = a_{2j}^{(0)} - m_{21} a_{1j}^{(0)}.$$

Notare che tutti gli elementi della riga cambiano, ma l'aspetto importante è aver ottenuto uno zero in posizione $(2, 1)$.

Ripetiamo questa trasformazione per tutte le righe. Quindi, lo stadio 1 del metodo consiste nel calcolare i moltiplicatori

$$m_{i1} = a_{i1}/a_{11}, \qquad i = 2, \ldots n$$

e modificare tutte le righe di $A^{(0)}$ come

$$A_i^{(1)} = A_i^{(0)} - m_{i1} A_1^{(0)} = \left[0, a_{i2}^{(1)}, \cdots, a_{in}^{(1)}\right], \qquad a_{ij}^{(1)} = a_{ij}^{(0)} - m_{i1} a_{1j}^{(0)}.$$

Otterremo la matrice $A^{(1)}$, che sarà fatta così,

$$A^{(1)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & & & & \\ 0 & a_{n2}^{(1)} & a_{n3}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix}.$$

E' facile vedere che da un punto di vista matriciale questa trasformazione è equivalente a

$$A^{(1)} = M_1 A, \qquad \text{dove} \quad M_1 = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -m_{21} & 1 & 0 & \cdots & 0 \\ -m_{31} & 0 & 1 & \cdots & 0 \\ \vdots & & & & \\ -m_{n1} & 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Nel secondo passaggio, annulliamo tutti gli elementi della colonna 2, sotto la diagonale. Dunque, calcoliamo i moltiplicatori $m_{i,2}$, $i \geq 3$. Ora la matrice dei moltiplicatori della seconda colonna sarà

$$A^{(2)} = M_2 A^{(1)}, \qquad \text{con} \quad M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & -m_{32} & 1 & 0 & \cdots & 0 \\ 0 & -m_{42} & 0 & 1 & \cdots & 0 \\ \vdots & & & & & \\ 0 & -m_{n2} & 0 & 0 & \cdots & 1 \end{bmatrix},$$

dove $m_{i2} = a_{i2}^{(1)}/a_{22}^{(1)}$, $i = 3, \ldots n$. Anche qui, bisogna avere $a_{22}^{(1)} \neq 0$, e, per il momento, supponiamo che questa condizione sia soddisfatta.

Se questa condizione non fosse soddisfatta, ovvero se $a_{22}^{(1)} = 0$, si procede nuovamente con una permutazione di righe: si scambia la riga 2 con una riga $i > 2$ tale che $a_{i2}^{(1)} \neq 0$. In generale, allo stadio k dell'algoritmo, se l'elemento pivot $a_{kk}^{(k-1)}$ è nullo (o numericamente troppo piccolo), si scambia la riga k con una delle righe successive per rendere possibile il passo di eliminazione.

Si noti che la moltiplicazione per M_2 lascia inalterate le prime 2 righe e la prima colonna di $A^{(1)}$, mentre trasforma in zeri gli elementi sulla seconda colonna, sotto la diagonale principale, e modifica tutti gli altri elementi. Otteniamo la matrice $A^{(2)}$

$$A^{(2)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2n}^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & & & & \\ 0 & 0 & a_{n3}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix}.$$

Ripetiamo il procedimento per tutte le colonne di A in modo da inserire zeri sotto la diagonale principale in ogni colonna. Allo stadio k avremo,

$$m_{ik} = a_{ik}^{(k-1)}/a_{k,k}^{(k-1)}, \quad i = k+1, \ldots n, \quad k = 1, \ldots n-1.$$

La matrice M_k è una matrice identità, con gli elementi sotto la diagonale principale della colonna k, $M_{k_{i,k}} = -m_{ik}$. Dunque, la matrice M_k modifica gli elementi $A_{ij}^{(k-1)}$ solo per $i, j \geq k+1$, e produce degli zeri in $A_{ik}^{(k-1)}$, $i = k+1, \ldots n$.

Il procedimento termina per $k = n - 1$, quando la matrice $A^{(n-1)}$ ha tutti gli elementi sotto la diagonale principale uguali a zero. Da un punto di vista matriciale,

$$U := A_{ij}^{(n-1)} = M_{n-1} \dots M_1 A.$$

Applicando a b le stesse modifiche apportate alla matrice A, si ottiene il sistema

$$Ux = \tilde{b} = M_{n-1} \dots M_1 b.$$

Nella pratica numerica, il pivoting non viene utilizzato solo per evitare divisioni per zero, ma anche per migliorare la stabilità numerica dell'algoritmo. Per questo motivo, è comune adottare il *pivoting parziale*, che consiste nello scegliere, ad ogni colonna k, come pivot l'elemento di massimo valore assoluto tra quelli appartenenti alla colonna k e alle righe $i \geq k$.

Algorithm 6 Metodo di eliminazione di Gauss con pivoting parziale

Require: A matrice $n \times n$ invertibile, b vettore $n \times 1$
Ensure: la matrice triangolare superiore U e il vettore $\tilde{b}$

for $k = 1 \dots n - 1$ **do**
  Scelta del pivot
  Trova l'indice $p \geq k$ tale che
$$|a_{pk}| = \max_{i=k,\dots,n} |a_{ik}|$$
  if $a_{pk} = 0$ **then**
    errore: la matrice non è invertibile
  end if
  if $p \neq k$ **then**
    Scambia la riga k con la riga p in A
    Scambia gli elementi b_k e b_p
  end if
  Eliminazione
  for $i = k + 1 \dots n$ **do**
    Calcola il moltiplicatore: $m_{ik} = a_{ik} / a_{kk}$
    for $j = k \dots n$ **do**
      $a_{ij} = a_{ij} - m_{ik}\, a_{kj}$
    end for
    $b_i = b_i - m_{ik}\, b_k$
  end for
end for
$U \leftarrow A, \quad \tilde{b} \leftarrow b$

Una volta trasformato il sistema $Ax = b$ nel sistema triangolare equivalente $Ux = \tilde{b}$, si applica il metodo di risoluzione all'indietro (vedi Algortimo 5).

15.4 Scelta tra metodi diretti e iterativi

La scelta tra metodi diretti e metodi iterativi dipende da alcune caratteristiche strutturali del problema lineare da risolvere.

Dimensione del sistema. I metodi diretti sono adatti a sistemi di dimensione piccola o moderata. Per sistemi di grandi dimensioni, in particolare quelli provenienti dalla discretizzazione

di problemi alle derivate parziali, il costo computazionale e la memoria richiesta da una fattorizzazione diventano rapidamente proibitivi, mentre i metodi iterativi mantengono un costo per iterazione ridotto e scalano meglio con la dimensione.

Sparsità della matrice. Le matrici sparse sono tipiche di molti problemi applicativi. Nei metodi diretti, durante la fattorizzazione si genera riempimento (*fill-in*) che aumenta il numero di elementi non nulli e di conseguenza il costo e la memoria. I metodi iterativi preservano la sparsità e richiedono solo prodotti matrice-vettore, risultando più efficienti nelle applicazioni su larga scala.

Accuratezza richiesta. Una fattorizzazione diretta fornisce una soluzione con accuratezza comparabile alla precisione macchina. I metodi iterativi producono una successione di approssimazioni controllata tipicamente tramite la norma del residuo o la variazione tra iterate successive, fornendo una soluzione tanto più accurata quanto maggiore è il numero di iterazioni.

Riutilizzo della fattorizzazione. Nel caso in cui si debbano risolvere più sistemi con la stessa matrice e termini noti differenti, la fattorizzazione diretta può essere riutilizzata, rendendo estremamente ridotto il costo dei sistemi successivi. Se invece la matrice varia frequentemente, come in schemi non lineari o problemi evolutivi, i metodi iterativi risultano più flessibili ed economici.

Requisiti di memoria. I metodi diretti richiedono memoria significativa, soprattutto per matrici grandi e prive di struttura particolare. I metodi iterativi richiedono tipicamente memoria dell'ordine di $O(n)$, limitata ai vettori necessari per l'iterazione, e rappresentano quindi una scelta obbligata nei problemi di dimensione elevatissima.

In conclusione, i metodi diretti sono indicati per sistemi piccoli o moderati, o quando è richiesta un'elevata accuratezza. I metodi iterativi rappresentano invece la scelta naturale per sistemi di grandi dimensioni, sparsi o caratterizzati da proprietà strutturali favorevoli alla convergenza.