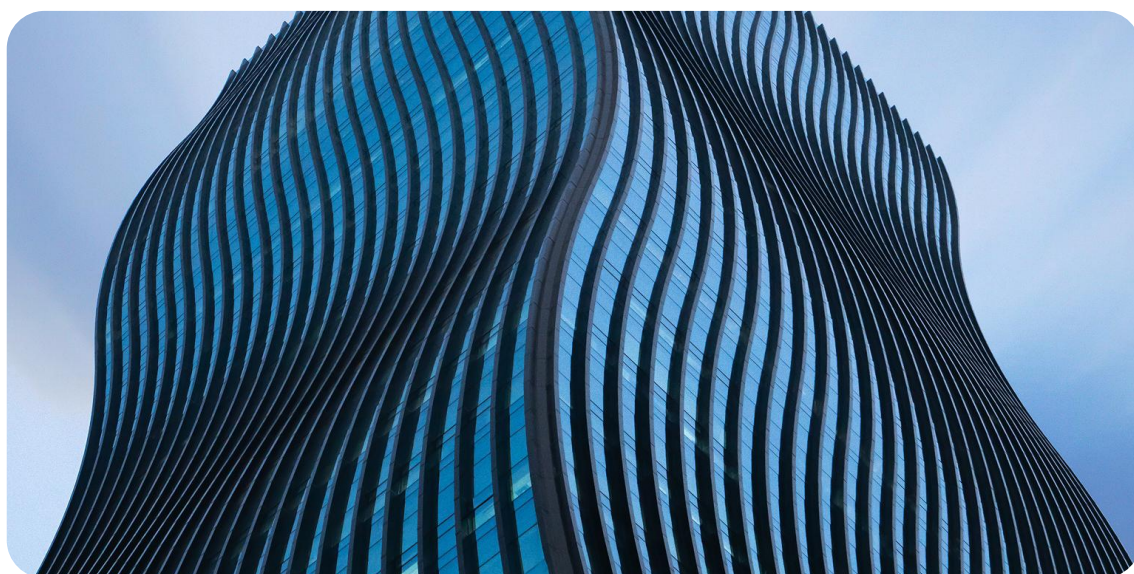


SAPIENZA, UNIVERSITY OF ROME
COURSE OF APPLIED COMPUTER SCIENCE AND ARTIFICIAL
INTELLIGENCE (ACSAI)
3RD YEAR, 2ND SEMESTER

ANALISI E CALCOLO NUMERICO



NOTES BY LEONARDO BIASON
COURSE TAUGHT BY PROF. DOMENICO VITULANO



SAPIENZA
UNIVERSITÀ DI ROMA

L

About these notes

Those notes were made during my three years of university at Sapienza, and **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

License

The decision of licensing this work was taken since these notes come from **university classes**, which are protected, in turn, by the **Italian Copyright Law** and the **University's Policy** (thus Sapienza Policy). By licensing these works I'm **not claiming as mine** the materials that are used, but rather the creative input and the work of assembling everything into one file. All the materials used will be listed here below, as well as the names of the professors (and their contact emails) that held the courses. The notes are freely readable and can be shared, but **can't be modified**. If you find an error, then feel free to contact me via the socials listed in my [website](#). If you want to share them, remember to **credit me** and remember to **not** obscure the **footer** of these notes.

Bibliography & References

- [1] S. C. Chapra, R. P. Canale. (2015) *Numerical Methods for engineers (Seventh edition)*, McGraw Hill

The "*Analisi e Calcolo Numerico*" course was taught in the Spring semester in 2025 by prof. Domenico Vitulano (domenico.vitulano@uniroma1.it)

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my [website](#). Good luck! Leonardo Biason
→ leonardo@biason.org

CHAPTER 1 ► INTRODUZIONE AL CALCOLO NUMERICO PAGE 1

1.1	Errori di approssimazione	2
1.2	La rappresentazione IEEE 754	4
1.3	Algoritmi e condizionamento dei problemi	8

CHAPTER 2 ► MATLAB PAGE 11

2.1	Variabili, handling della memoria e formati	13
2.2	Tipi di dati	15
2.2.1	Vettori, matrici e tensori	15
2.2.2	Celle	17
2.2.3	Structs	18
2.3	Funzioni	18
2.3.1	Funzioni particolari	19
2.4	Costrutti di flow control	20

CHAPTER 3 ► SISTEMI DI EQUAZIONI NON LINEARI PAGE 21

3.1	Metodo di bisezione (o dicotomico)	23
3.1.1	Ordine di convergenza e criteri di arresto	25
3.2	Metodo di Newton-Raphson	26

CAPITOLO 1

Introduzione al calcolo numerico

Grazie al costante sviluppo dei computers negli scorsi decenni, la comunità scientifica ha avuto modo di usufruire di strumenti di calcolo sempre più precisi e complessi, necessari per risolvere alcuni problemi di vario tipo. Questo sviluppo ha visto anche un cospicuo interesse verso i metodi di calcolo numerico, che permettono di risolvere in modo non-analitico problemi specifici che non sarebbero, altrimenti, risolvibili. Infatti, seppur non esista sempre una soluzione analitica, **esiste sempre una soluzione numerica** per un modello matematico **ben posto** e **condizionato**, che assuma tuttavia certe assunzioni del corrispondente modello fisico.

La realtà infatti non è sempre modellabile attraverso semplici formule fisiche: a volte ci sono parecchie variabili da tenere in conto quando si cerca di risolvere un problema, e non è sempre plausibile considerare tutte queste variabili assieme, soprattutto se il problema va risolto senza l'assistenza di un calcolatore. Consideriamo il seguente esempio: un giocatore di golf colpisce una pallina con una certa velocità U , e noi vogliamo sapere per quale angolo α la distanza che verrebbe percorsa dalla pallina da golf sarebbe massima prima che quest'ultima tocchi terra.

Grazie alla seconda legge di Newton, possiamo calcolare la distanza percorsa dalla pallina, e se trascurassimo la resistenza dell'aria sarebbe abbastanza semplice trovare la soluzione analitica. Tuttavia, considerando questa resistenza, le equazioni del moto si complicano notevolmente, e determinare la soluzione analitica diventa ora impossibile. Tuttavia, la soluzione numerica rimane calcolabile attraverso l'impiego di metodi numerici adatti.

È comunque importante considerare anche il tipo di modello utilizzato: in base al modello matematico di partenza e al metodo utilizzato, si possono ottenere risultati diversi. L'importante è saper scegliere il metodo giusto e l'approssimazione migliore del modello.

Per il calcolo numerico, la risoluzione di un problema avviene attraverso i seguenti steps:

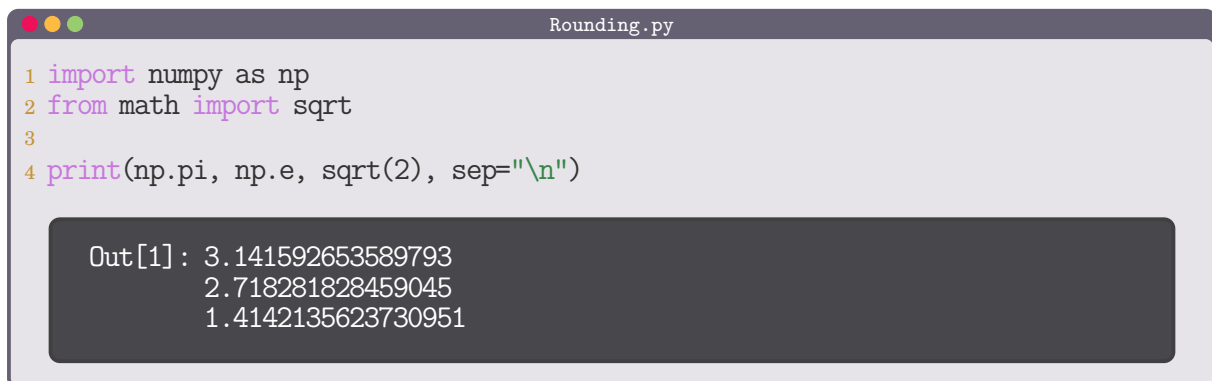
- formulare un **modello matematico** in base al problema dato, che diventi uno schema per definire il metodo numerico e l'algoritmo di soluzione;
- scegliere un **metodo numerico** che aiuti nella risoluzione del problema;
- definire un **algoritmo** che porti alla soluzione desiderata;
- analizzare la **soluzione numerica** e interpretarla, capendo se quest'ultima sia una valida soluzione o meno. Si dice che una soluzione numerica sia **accettabile** se e solo se sia possibile **stimare gli errori** che accompagnano la soluzione stessa.

SEZIONE
1.1

Errori di approssimazione

Quando si calcola una soluzione numerica, ci sono varie, possibili fonti di errori che possono condizionare il risultato finale. È possibile avere errori di **misura** (dati dalla precisione dello strumento), **inerenti** (creati da un'eccessiva semplificazione del modello reale), di **troncamento** (generati da una discretizzazione del risultato, generalmente presenti quando si usano metodi numerici che richiedono convergenza), e di **arrotondamento** (creati dalla macchina che performa i calcoli, in quando la precisione è sempre limitata).

Ogni computer dispone di un sistema numerico piuttosto primitivo: questo infatti dispone di un sistema **finito** di numeri, la cui lunghezza è anch'essa **finita**. Se normalmente, in campi analitici, siamo abituati a pensare con un insieme di numeri infinito (come quello dei numeri reali, \mathbb{R}), con i computer, quando si performano calcoli di analisi numerica, si considera un insieme ristretto, detto dei **numeri macchina** \mathbb{F} . Consideriamo ad esempio alcune delle costanti più famose nel mondo matematico: π , e e $\sqrt{2}$. Noi sappiamo che questi numeri sono irrazionali, e che si espandono all'infinito. Proviamo a chiedere a una macchina di dirci quali sono questi numeri. Eseguendo il seguente script di Python, otterremo il seguente risultato:



```

1 import numpy as np
2 from math import sqrt
3
4 print(np.pi, np.e, sqrt(2), sep="\n")

```

Out[1]: 3.141592653589793
2.718281828459045
1.4142135623730951

Noi sappiamo che in realtà questi numeri si estendono molto più in profondità di quello che ci ha ritornato Python. Infatti:

- $\pi = 3,1415926535897932384626433\dots$;
- $e = 2,71828182845904523536\dots$;
- $\sqrt{2} = 1,4142135623730950488\dots$

Qua notiamo già uno dei primi errori che si incontra quando si usa un calcolatore: i numeri sono **arrotondati** ad una certa cifra. L'arrotondamento genera spesso qualche tipo di errore, ma è necessario che i numeri subiscano una procedura di arrotondamento prima di poter essere usati da un calcolatore, poiché altrimenti non entrerebbero nella memoria di quest'ultimo, che ricordiamo essere limitata.

Errore di arrotondamento

DEFINITION

Definiamo l'**errore di arrotondamento** come la **differenza** tra il **numero reale** $x \in \mathbb{R}$ e il **numero macchina** $m \in \mathbb{F}$ corrispondente:

$$e_{\text{arr}} = x - m$$

Se un calcolatore approssima tutti i numeri alla *Desima* cifra decimale, allora diciamo che l'errore di arrotondamento è compreso nell'**intervallo** $[-0,5 \cdot 10^{-D}, +0,5 \cdot 10^{-D}]$.

Riguardo lo scopo di queste note: MATLAB verrà usato durante il corso per implementare certi metodi numerici. Tale linguaggio di programmazione lavora con 15 cifre decimali significative. Fino a quando non verrà introdotto MATLAB tuttavia, verrà usato Python, che ne usa fino a 17 (anche se negli esempi precedenti π e e hanno usato solo 15 cifre, probabilmente a causa del pacchetto numpy).

Consideriamo un esempio per comprendere l'importanza degli errori di arrotondamento:

1.1.1

Si considerino le due funzioni seguenti, che sono algebricamente equivalenti:

$$q_1(x) = (x - 1)^7 \quad q_2(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

Vogliamo calcolare il valore numerico di $q_1(x)$ e $q_2(x)$ con due valori di x , ovvero sia 1 e 1,0001, e confrontare il loro valore esatto con l'errore di arrotondamento. Vogliamo inoltre usare una macchina che lavori con 15 cifre significative. Usando il seguente script in Python, possiamo ottenere i nostri risultati:

```

ApproxExample.py
1 from math import pow
2
3 def q1(x) -> float:
4     return (x - 1) ** 7
5
6 def q2(x) -> float:
7     return pow(x, 7) - 7 * pow(x, 6) + 21 * pow(x, 5) \
8         - 35 * pow(x, 4) + 35 * pow(x, 3) - 21 * pow(x, 2) + 7 * x - 1
9
10 def rounding_error(real, machine) -> float:
11     return float(real) - float(machine)
12
13 # La lista contiene tuple del tipo (x, valore_reale)
14 for i, expect in [(1, 0), (1.0001, 10**(-28))]:
15     # Approssimazione del numero alla 10a cifra decimale
16     res1, res2 = "{0:.10g}".format(q1(i)), "{0:.10g}".format(q2(i))
17     err1, err2 = rounding_error(expect, res1), rounding_error(expect,
18         res2)
19     print(f"With x = {i}, expect {expect}\nQ1 = {res1} | E1 = {err1}\n
        nQ2 = {res2} | E2 = {err2}\n")

```

```

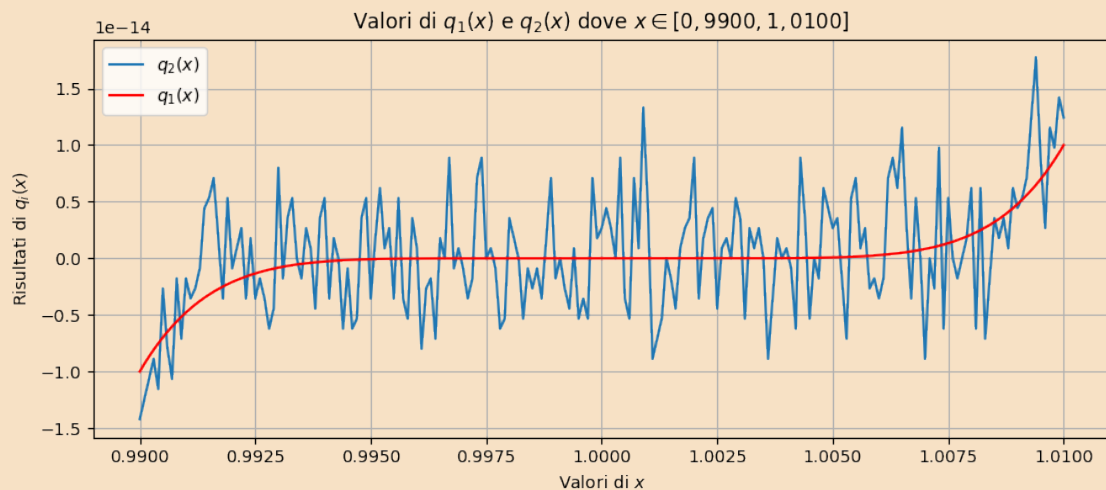
Out[1]: With x = 1, expect 0
        Q1 = 0 | E1 = 0.0
        Q2 = 0 | E2 = 0.0

        With x = 1.0001, expect 1e-28
        Q1 = 1e-28 | E1 = 0.0
        Q2 = 1.776356839e-15 | E2 = -1.7763568389999e-15

```

Come possiamo notare dall'output dello script, con $x = 1$ non c'è alcuna differenza tra $q_1(x)$ e $q_2(x)$, ma con $x = 1,0001$ iniziano ad esserci le prime differenze. Infatti, nel secondo caso abbiamo un errore di arrotondamento di circa $-1,78 \cdot 10^{-15}$. Questo semplice esempio dimostra come due quantità che sono algebricamente uguali possono in realtà portare a due risultati numerici completamente diversi.

Questo comportamento può essere inoltre osservato attraverso il seguente grafico:



Notiamo infatti che, mentre $q_1(x)$ ha un comportamento più lineare, $q_2(x)$ è molto più scabro, e questo accade proprio a causa degli errori di approssimazione.

SEZIONE 1.2

La rappresentazione IEEE 754

Ogni numero reale x può essere espresso come una sequenza di infinite cifre, e tale sequenza dipende dalla **base di rappresentazione** β . Di norma, la base con cui noi esseri umani facciamo calcoli è $\beta = 10$.

Qualsiasi cifra può essere espressa in qualsiasi base. Per farlo, faremo un esempio con π . Il numero infatti può essere scritto come segue:

$$\pi = 3,14159... = \frac{3}{10^0} + \frac{1}{10^{-1}} + \frac{4}{10^{-2}} + \frac{1}{10^{-3}} + \frac{5}{10^{-4}} + \frac{9}{10^{-5}} + \dots$$

L'idea è che per esprimere un qualsiasi numero x in una base β , possiamo scrivere il numero come

$$x = x_m \cdot \beta^m + x_{m-1} \cdot \beta^{m-1} + \dots + x_1 \cdot \beta^1 + x_0 \cdot \beta^0 + x_{-1} \cdot \beta^{-1} + \dots + x_{-m} \cdot \beta^{-m}$$

dove $0 \leq x_i \leq \beta - 1$, $\forall i \in [m, -m]$.

Sappiamo che i computer funzionano in codice binario, quindi non possono interpretare i numeri in base decimale come facciamo noi umani. Per poter far sì che un computer riconosca un numero, questo va prima convertito in base 2. Ci sono vari modi per rappresentare un numero in binario: che sia con o senza segno, a virgola mobile o meno...

C'è tuttavia uno standard che i computer adottano, che è stato sviluppato dall'IEEE, che viene usato per rappresentare tutti i numeri in binario, che abbiano un segno o che siano a virgola mobile: l'**IEEE 754**.

Per questo standard, ogni numero può essere espresso nella seguente rappresentazione:

$$x = \underbrace{\pm}_{\text{Segno}} \underbrace{(1 + a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + a_{-3} \cdot 2^{-3} + \dots + a_{-m} \cdot 2^{-m})}_{\text{Mantissa normalizzata}} \cdot \underbrace{2^e}_{\text{Esponente}}$$

s	e	m
-----	-----	-----

dove il segno viene rappresentato con 1 bit, la mantissa con m bits e l'esponente con n bits. Generalmente i numeri in IEEE 754 si possono esprimere con 16 (precisione dimezzata), 32 (singola precisione) o 64 bits (precisione doppia). Segue una tabella che segna quanti bits vengono assegnati ad ogni formato:

Formato	Segno	Esponente	Mantissa	Bias	Numero totale di bits
Precisione mezza	1	5	10	15	16
Singola precisione	1	8	23	127	32
Doppia precisione	1	11	52	1023	64

Dato che i computer hanno una precisione finita, limitata a p cifre, è chiaramente impossibile per questi rappresentare numeri che abbiano più di p cifre. Per poter rappresentare numeri con più cifre, è necessario **arrotondare** il numero. L'arrotondamento può avvenire in due modi: o tramite **troncamento** o tramite **arrotondamento simmetrico**.

Troncamento e Arrotondamento simmetrico

Per **troncamento** si definisce quell'operazione per cui un numero a n cifre viene rappresentato come un numero a p cifre, dove le ultime $n - p$ cifre sono uguali a 0:

$$x^* = \text{trunc}(x) \implies x_{-k} = 0, \forall k \geq p$$

Per **arrotondamento simmetrico** si definisce un'operazione di troncamento su un numero x a cui può essere aggiunta un'unità alla cifra x_{-p+1} se la cifra x_{-p} è maggiore o uguale di $\frac{\beta}{2}$:

$$x^* = \text{trunc}(x + 0,5 \cdot \beta^{-p+1} \cdot \beta^e) \implies \begin{cases} x_{-p+1} = x_{-p+1} & \text{se } x_{-p} < \frac{\beta}{2} \\ x_{-p+1} = x_{-p+1} + 1 & \text{se } x_{-p} \geq \frac{\beta}{2} \end{cases}$$

Arrotondare comporta sempre la presenza di un errore, e tali errori **non possono essere trascurati**, in quanto possono potenzialmente alterare il risultato finale in modi disastrosi. Un esempio è il caso del processore Intel Pentium (1994), che portava a risultati imperfetti a causa dell'arrotondamento dei numeri alla quinta cifra decimale.

Un altro esempio di errore dato dall'arrotondamento è la **cancellazione numerica**. Per spiegare meglio questo fenomeno, consideriamo il seguente esempio:

1.2.1

EXAMPLE

Considerando un'equazione di secondo grado del tipo $ax^2 + bx + c = 0$, vogliamo calcolare le radici dell'equazione dati i valori di a , b e c . Vogliamo calcolare x_1 e x_2 sia attraverso la formula classica delle radici (quindi $x = \frac{-b \pm \sqrt{\Delta}}{2a}$), sia attraverso una forma più compatta:

$$x_1 = \frac{2c}{-b + \sqrt{\Delta}} \quad x_2 = \frac{c}{ax_1}$$

Per farlo, consideriamo il seguente script di Python, che con la funzione `solve_f()` calcola le due radici con la formula classica e che con la funzione `solve_f_alt()` calcola invece le due radici con le formule alternative sopra menzionate:

```
NumericalAbsorption.py
1 def solve_f(a, b, c) -> tuple[float, float]:
2     delta = pow(b, 2) - 4 * a * c
3     x1, x2 = (-b - sqrt(delta)) / (2 * a), (-b + sqrt(delta)) / (2 * a)
4     return (x1, x2)
5
6 def solve_f_alt(a, b, c) -> tuple[float, float]:
7     delta = pow(b, 2) - 4 * a * c
8     x1 = (2 * c) / (-b + sqrt(delta))
9     x2 = c / (a * x1)
10    return (x1, x2)
11
12 def f(a, b, c, x) -> float:
13     return a * pow(x, 2) + b * x + c
14
15
16 inputs = [[1, 4, 3], [1, -206.5, 0.01021]]
17 for a, b, c in inputs:
18     x1, x2 = solve_f(a, b, c)
19     print(f"With a = {a}, b = {b}, c = {c}\n x1 = {x1}\n x2 = {x2}\n
20           f({x1}) = {f(a, b, c, x1)}\n f({x2}) = {f(a, b, c, x2)}\n")
21
22     x1, x2 = solve_f_alt(a, b, c)
23     print(f"Second method\n x1 = {x1}\n x2 = {x2}\n f({x1}) = {f(a,
24           b, c, x1)}\n f({x2}) = {f(a, b, c, x2)}\n\n")
```

Raccogliamo gli output della prima parte del codice nella seguente tabella, in cui mostriamo i risultati ottenuti con la funzione `solve_f()`:

a, b, c	x_1	x_2	$f(x_1)$	$f(x_2)$
1, 3, 4	-3	-1	0	0
1, -206,5, 0,01021	$4,944... \cdot 10^{-5}$	206,499...	$5,454... \cdot 10^{-13}$	$-3,702... \cdot 10^{-13}$

In questa seconda tabella invece, raccogliamo i risultati ottenuti grazie alla funzione `solve_f_alt()`:

EXAMPLE

a, b, c	x_1	x_2	$f(x_1)$	$f(x_2)$
1, 3, 4	-3	-1	0	0
1, -206,5, 0,01021	$4,944... \cdot 10^{-5}$	206,499...	$1,734... \cdot 10^{-18} \simeq 0_m$	$-3,702... \cdot 10^{-13}$

Sebbene per il primo set di inputs (quindi con $a = 1$, $b = 3$ e $c = 4$) i risultati siano gli stessi, con il secondo set i risultati iniziano ad essere diversi da quel che ci aspetteremmo. Infatti, il risultato di $f(x_1)$ e $f(x_2)$ dovrebbe essere uguale a 0, eppure è sempre un numero abbastanza vicino allo zero (nella seconda tabella, il risultato di $f(x_1)$ per il secondo set di inputs è infatti segnato come simile allo zero macchina, 0_m).

Ancora più interessante è il risultato di x_1 , quando viene usato il secondo set di inputs: il valore infatti è dato da $-b - \sqrt{\Delta}$, che, dati i nostri inputs, corrisponde al calcolo della differenza tra b e $\sqrt{\Delta}$. Questi due numeri però sono molto vicini fra di loro, e la loro differenza è un esempio di cancellazione numerica.

Realmente, la loro differenza dovrebbe risultare in un numero infinitesimamente piccolo, ma il computer lo approssima a 0 per impossibilità di immagazzinare numeri infinitesimamente piccoli.

Un errore non è mai fine a sé stesso: è possibile (talvolta certo) che si propaghi e che influenzi i risultati delle operazioni future. Definiamo qui i vari tipi di errori che si creano in base all'operazione che viene performata:

Errori di propagazione

DEFINITION

Si consideri come $fl(n)$ la rappresentazione in virgola mobile arrotondata del numero n , e si denoti con e_n l'errore corrispondente, cosicché:

$$e_n = \frac{fl(n) - n}{n} \implies fl(n) = n \cdot e_n + n = n \cdot (1 + e_n)$$

Si considerino due numeri x e y , e le loro rispettive rappresentazioni in virgola mobile. Definiamo i seguenti errori:

- **errore del prodotto** e_{xy} :

$$fl(x) \cdot fl(y) = (x \cdot (1 + e_x)) \cdot (y \cdot (1 + e_y)) = xy \cdot \underbrace{(1 + e_x + e_y + e_x \cdot e_y)}_{e_{xy}} \simeq xy \cdot \underbrace{(1 + e_x + e_y)}_{e_{xy}}$$

- **errore della divisione** $e_{\frac{x}{y}}$:

$$\frac{fl(x)}{fl(y)} = \frac{x \cdot (1 + e_x)}{y \cdot (1 + e_y)} = \frac{x}{y} \cdot (1 + e_x) \cdot (1 - e_y + e_y^2 + \dots) \simeq \frac{x}{y} \cdot (1 + \underbrace{e_x - e_y}_{e_{\frac{x}{y}}})$$

- **errore della somma** e_{x+y} :

$$fl(x) + fl(y) = x \cdot (1 + e_x) + y \cdot (1 + e_y) \simeq x + y + xe_x + ye_y =$$

$$= (x+y) \cdot \left(1 + \underbrace{\frac{x}{x+y} \cdot e_x + \frac{y}{x+y} \cdot e_y}_{e_{x+y}} \right)$$

In quest'ultimo caso, se $x, y > 0$, allora $|e_{x+y}| \leq |e_x| + |e_y|$; se $x, y < 0$, allora le quantità $\left| \frac{x}{x+y} \right|$ e $\left| \frac{y}{x+y} \right|$ possono essere molto grandi

SEZIONE 1.3

Algoritmi e condizionamento dei problemi

Fin dall'antica Grecia c'è sempre stata varia ambiguità su cosa fosse un algoritmo e su come definirlo, nonostante ci fosse sempre stata un'intuizione. È solo grazie alla tesi di Church-Turing nel 1936, che fu possibile definire formalmente cosa fosse un algoritmo (definito nella tesi come *metodo efficace*):

"Un metodo efficace è un metodo per cui ogni istruzione è precisamente pre-determinata, che è certo di produrre un output entro un numero finito di istruzioni"

~ Church-Turing

Possiamo tuttavia riformulare la tesi di Church-Turing per definire cosa sia un algoritmo, così da adattarla anche per i nostri scopi:

Algoritmo

Un **algoritmo** è una successione di **istruzioni finita** e **non ambigua**, che consente di ottenere risultati numerici a partire dai dati di input

Gli algoritmi che formuleremo attraverso queste note verranno implementate su un computer tramite un linguaggio di programmazione. Tutte le istruzioni sono operazioni logiche o aritmetiche, che vengono assegnate al computer seguendo la sintassi del linguaggio che useremo.

Come abbiamo visto in precedenza, la propagazione di un errore può avere effetti disastrosi, anche se l'errore è molto "piccolo". Se l'errore dovesse amplificarsi ad alti livelli, il risultato ottenuto non sarebbe affidabile. In questo caso, diremmo che l'algoritmo è **instabile**. Se invece gli errori di arrotondamento non vengono amplificati durante i calcoli, allora diciamo che l'algoritmo è **stabile**.

1.3.1

Data la funzione $f(x) = x^2 + 2px - q$, con $p^2 + q \geq 0$, vogliamo calcolare la radice di valore maggiore. Questa è data dalla seguente equazione

$$y = -p + \sqrt{p^2 + q}$$

Proviamo a creare uno script di Python che esegua questa equazione e che calcoli anche l'errore tra la soluzione del computer e quella prevista (usando la funzione `rounding_error()` dall'esempio 1.1.1)

```

AlgorithmStability-Part1.py
1 P = 1000
2 Q = 0.0180000000081
3 SOLUTION = 0.9 * pow(10, -5)
4
5 def solve_r1(p, q) -> float:
6     return -p + sqrt(pow(p, 2) + q)
7
8 r1 = solve_r1(P, Q)
9
10 print(f"[Algorithm 1] y = {r1}\n[Algorithm 1] Error = {rounding_error(
    SOLUTION, r1)}")

Out[1]: [Algorithm 1] y = 8.999999977277184e-06
        [Algorithm 1] Error = 2.2722815857102556e-14

```

Come possiamo notare, l'errore è abbastanza cospicuo, e per questo l'algoritmo è detto instabile. Questo perché, per $p \gg q$, la sottrazione tra p e $\sqrt{p^2 + q}$ comporta la cancellazione numerica. Proviamo invece con un altro algoritmo, che cerca di arrivare allo stesso risultato senza usare la sottrazione, così da essere stabile:

$$y = -p + \sqrt{p^2 + q} = \left(-p + \sqrt{p^2 + q}\right) \cdot \frac{(p + \sqrt{p^2 + q})}{(p + \sqrt{p^2 + q})} = \frac{q}{(p + \sqrt{p^2 + q})}$$

```

AlgorithmStability-Part2.py
1 def solve_r2(p, q) -> float:
2     return q / (p + sqrt(pow(p, 2) + q))
3
4 r2 = solve_r2(P, Q)
5
6 print(f"[Algorithm 2] y = {r2}\n[Algorithm 2] Error = {rounding_error(
    SOLUTION, r2)}")

Out[1]: [Algorithm 2] y = 9e-06
        [Algorithm 2] Error = 0.0

```

Non solo gli algoritmi possono essere stabili o instabili, ma in base a come vengono posti i problemi ci può essere più o meno suscettibilità alle perturbazioni dei dati. Consideriamo ad esempio il seguente problema: abbiamo una funzione $f : \mathbb{R} \mapsto \mathbb{R}$, e vogliamo calcolarne il valore y in un generico punto $x \in \mathbb{R}$. Normalmente avremmo che $x \rightarrow f(x) \rightarrow y$.

Ottenuto y , vogliamo misurare quale effetto produca una perturbazione $\Delta x = x^* - x$ (dove

x^* è un valore a nostra scelta) durante il calcolo di y . Possiamo inoltre riconsiderare Δx come il seguente: $\Delta y = y^* - y = f(x^*) - f(x)$

Usando lo sviluppo in serie di Taylor fino al primo ordine, possiamo riscrivere Δy come segue:

$$\Delta y = f(x^*) - f(x) = f'(x) \cdot \underbrace{\Delta x}_{x^* - x}$$

Dividendo ciò che abbiamo ottenuto fino ad ora per y , possiamo ricavare l'**errore relativo**:

$$\left| \frac{\Delta y}{y} \right| \simeq \left| \frac{f'(x)}{f(x)} \right| \cdot |\Delta x| = \underbrace{\left| \frac{f'(x) \cdot x}{f(x)} \right|}_{C_P} \cdot \left| \frac{\Delta x}{x} \right|$$

Errore relativo

DEFINITION

L'**errore relativo** e_r è l'errore che c'è tra un valore x^* e un valore comparato x , ed è sempre **espresso in percentuale**. Tale errore si calcola come segue:

$$e_r = \frac{|x^* - x|}{x}$$

Nel calcolo precedente abbiamo evidenziato una parte dell'equazione, chiamandola C_P . Tale numero è importante per noi, e si chiama **numero di condizionamento del problema**.

Numero di condizionamento del problema

DEFINITION

Il **numero di condizionamento del problema** C_P è un numero che determina se un dato problema è **malcondizionato** (ovverosia che a **piccole perturbazioni** dei dati corrispondono **grandi variazioni** dei risultati) o **ben condizionato**. Se C_P è **grande**, allora il problema è **malcondizionato**, altrimenti è ben condizionato.

$$C_P = \left| \frac{f'(x) \cdot x}{f(x)} \right|$$

Sebbene la propagazione dell'errore dipenda dall'algoritmo, il condizionamento del problema non dipende né dall'algoritmo, né dagli errori generati. Infatti, il **condizionamento dipende solo ed unicamente dal problema** e dai dati di input. Se il problema è molto sensibile, e quindi malcondizionato, alle variazioni di input, allora **non esiste nessun algoritmo** che riesca a ritornare una soluzione stabile al problema.

CAPITOLO MATLAB 2

MATLAB è un linguaggio di programmazione sviluppato negli anni '70, che viene usato per sviluppare modelli matematici, svolgere simulazioni e analisi dei dati. Il modo in cui MATLAB funziona è detto **interattivo**, poiché viene tutto eseguito nella console. È possibile anche eseguire più comandi assieme nello stesso prompt, separando tutti i comandi con delle virgole. Ad esempio:

```
Terminal
>> 2+3, 7*2, 9+1*3

ans = 5
ans = 14
ans = 12
```

Operazioni più lunghe possono essere scritte su più righe usando i "...". Gli operatori disponibili sono i seguenti:

Operazione	Operatore
Somma	+
Sottrazione	-
Moltiplicazione	*
Divisione	/
Potenza	^
Minore	<
Maggiore	>
Minore o uguale	<=
Maggiore o uguale	>=
Uguale	==
Diverso	~=

In MATLAB, è possibile anche usare gli operatori logici, quali l'AND, l'OR e il NOT. Chiaramente, anche i gate logici più complessi, che vengono costruiti con gli operatori logici più semplici, sono disponibili.

Operazione	Operatore
AND	&
OR	
NOT	~

Ci sono anche alcune costanti, che vengono incluse in MATLAB di default dalla libreria standard. Qui alcune di queste vengono elencate:

Operazione	Operatore
Infinito (∞)	inf
π	pi
i	i
Numero massimo rappresentabile	realmax
Numero minimo rappresentabile	realmin
Precisione della macchina	eps
Forma indeterminata / Not A Number	nan

Nella scorsa tabella, i valori `realmax` e `realmin` si riferiscono rispettivamente al valore massimo e minimo rappresentabile considerando numeri in IEEE 754 a doppia precisione (dunque a 64 bits). La libreria standard di MATLAB possiede varie funzioni matematiche, tra cui `sin(x)`, `cos(x)`, `tan(x)`, `log(x)`, etc... Nel caso in cui si volessero avere più informazioni circa una funzione, si può usare la funzione `help <funzione>`, dove `<funzione>` è la funzione di cui vogliamo ottenere più informazioni. Ad esempio:

```
Terminal
>> help log

log - Natural logarithm
This MATLAB function returns the natural logarithm ln(x) of each element
in array X.

Syntax
    Y = log(X)

Input Arguments
    X - Input array
         scalar | vector | matrix | multidimensional array | table |
         timetable

Output Arguments
    Y - Logarithm values
         scalar | vector | matrix | multidimensional array | table |
         timetable

Examples
    Natural Logarithm of Negative Number

See also log1p, log2, log10, exp, logm, reallog, loglog, semilogx,
    semilogy

Introduced in MATLAB before R2006a
Documentation for log
Other uses of log
```

Se dovessimo aver bisogno di una funzione che svolga un certo compito, ma non ci dovessimo ricordare qual'è la funzione adatta, possiamo usare invece la funzione `lookfor <keywords>`, dove `<keywords>` è un insieme di keywords per identificare la funzione che cerchiamo. Ad esempio:


```

Terminal
>> lookfor square

cgs          - Solve system of linear equations - conjugate
              gradients squared method
deconv       - Least-squares deconvolution and polynomial
              division
hypot        - Square root of sum of squares (hypotenuse)
lscov        - Least-squares solution in presence of known
              covariance
lsqminnorm   - Minimum norm least-squares solution to
              linear equation
lsqnonneg    - Solve nonnegative linear least-squares
              problem
lsqr         - Solve system of linear equations -
              least-squares method
[...]
```

SEZIONE 2.1

Variabili, handling della memoria e formati

Le variabili su MATLAB vengono assegnate e dichiarate similmente a Python: l'assegnazione e la dichiarazione avvengono allo stesso momento. Ad esempio, se volessimo dichiarare la variabile $a = 4$ ci basterebbe eseguire il seguente codice:

```

Terminal
>> a = 4

a = 4
```

Possiamo visualizzare il contenuto di una variabile in due modi: o chiamando la variabile nella console, o usando la funzione `disp(<variabile>)`. Segue un esempio:

```

Terminal
>> a
>> disp(a)

a = 4

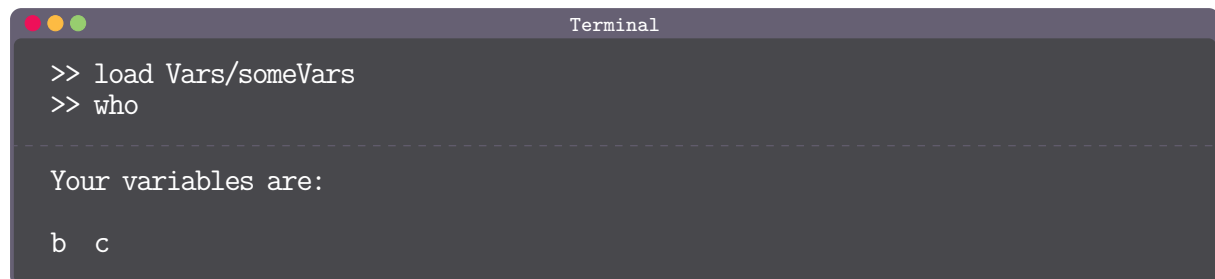
4
```

Per cancellare tutte le variabili dalla memoria si usa il comando `clear`. Per salvarne alcune tra una sessione e l'altra, possiamo usare il comando `save <filename> [<var1> <var2> ...]`, dove `<filename>` è il file in cui salveremo le variabili (in estensione `.mat`), mentre `[<var1> <var2> ...]` è una lista di variabili che vogliamo salvare. Ad esempio:

```

Terminal
>> b = 5;
>> c = 7;
>> save Vars/someVars b c
```

Una volta cancellate le variabili dalla memoria, usando il comando `who` non le vedremo più. Possiamo caricare nuovamente le variabili all'interno di MATLAB usando il comando `load <filename>`, che caricherà tutte le variabili all'interno del file `<filename>`. Non è necessario includere l'estensione `.mat`.



```
Terminal
>> load Vars/someVars
>> who

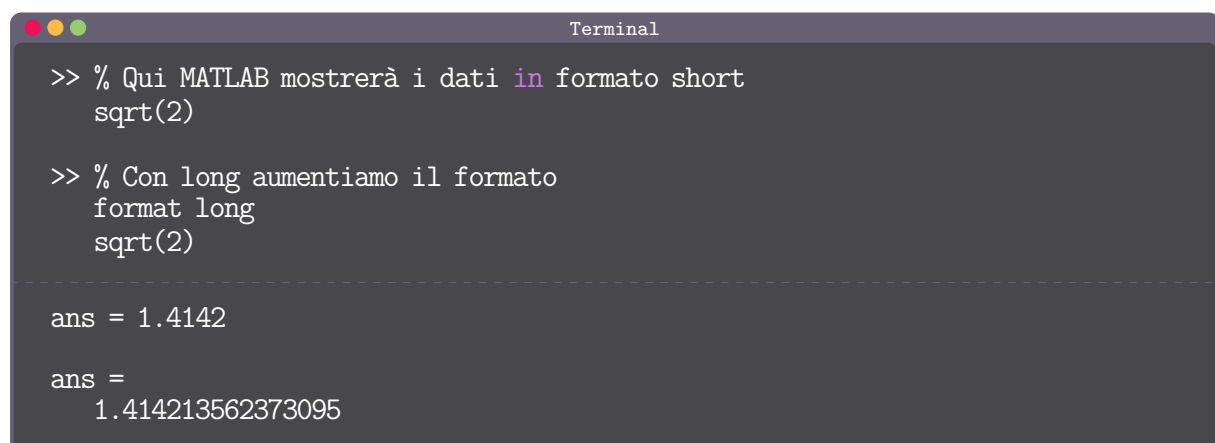
Your variables are:

b  c
```

MATLAB ha vari formati per i dati, simili concettualmente ai tipi di Python, molto vicini ai tipi di C. Ad esempio:

Formato	Descrizione
double	Numeri in doppia precisione
uint8	Interi senza segno a 8 bits
uint16	Interi senza segno a 16 bits
uint32	Interi senza segno a 32 bits
int8	Interi con segno a 8 bits
int16	Interi con segno a 16 bits
int32	Interi con segno a 32 bits
single	Numeri a singola precisione
char	Caratteri, 2 bytes per carattere
logical	Valore che è 0 o 1, generalmente usato come valore Booleano

Possiamo impostare anche un formato di visualizzazione dei dati tramite la funzione `format`. Tuttavia, questo formato sarà valido **solo per la visualizzazione dei dati**: all'interno di MATLAB i calcoli verranno effettuati con la stessa precisione standard di MATLAB. Facciamo un esempio:



```
Terminal
>> % Qui MATLAB mostrerà i dati in formato short
sqrt(2)

ans = 1.4142

>> % Con long aumentiamo il formato
format long
sqrt(2)

ans =
1.414213562373095
```

Se volessimo cambiare il tipo dei dati, possiamo farlo usando i costruttori dei vari tipi. Ad esempio:

```

Terminal
>> a = 43.97;

>> int16(a)
>> double(a)
>> uint8(a)

-----
ans = int1644
ans = 43.9700
ans = uint844

```

SEZIONE 2.2 Tipi di dati

Oltre ai formati di numeri menzionati fino ad ora, MATLAB ha anche altri tipi di dati, che ritornano comodi per esprimere costrutti matematici come vettori, matrici e tabelle.

2.2.1 Vettori, matrici e tensori

In MATLAB è possibile usare vettori, matrici e in generale tensori a n dimensioni. Come in C e in Python, vettori e matrici sono realizzabili tramite array a rispettivamente una e due dimensioni. Anche le variabili in realtà sono considerate internamente come arrays: infatti uno scalare è rappresentato tramite matrici a dimensione 1×1 . Possiamo realizzare vettori riga e vettori colonna, in base al carattere usato per separare i valori:

- usando la virgola , (o degli spazi) possiamo creare vettori riga;
- usando il punto e virgola ; possiamo creare vettori colonna.

```

Terminal
>> A = [10, 20, 30, 40, 50]
>> B = [10; 20; 30; 40; 50]

-----
A = 1x5
    10    20    30    40    50

B = 5x1
    10
    20
    30
    40
    50

```

Per ottenere il trasposto di un vettore o di una matrice, si usa il simbolo ':

```

Terminal
>> A'

```

```

ans = 5x1
    10
    20
    30
    40
    50

```

Per ottenere invece il valore posto in una certa posizione di un vettore, si usa la seguente notazione: `vettore(posizione)`. In MATLAB, **gli indici non partono da 0, ma da 1**. Possiamo usare anche una notazione simile allo slicing di Python. Infatti, esiste una notazione più complessa, che è `vettore(inizio:passo:fine)`, dove `inizio` determina l'indice di inizio della sequenza che vogliamo esprimere, `passo` indica ogni quanti elementi serve prendere un valore, `fine` (**inclusivo**) indica quando fermarsi con lo slice. Ad esempio:

```

>> A(1:2:4)

ans = 1x2
    10    30

```

Con questa notazione possiamo anche creare nuovi vettori. Ad esempio:

```

>> C = [0.9:0.01:1.1]

C = 1x21
    0.9000    0.9100    0.9200    0.9300    0.9400    0.9500    0.9600
    0.9700    0.9800    0.9900    1.0000    1.0100    1.0200    1.0300
    1.0400    1.0500    1.0600    1.0700    1.0800    1.0900    1.1000

```

Contrariamente a C, la dimensione e i valori di un array possono essere modificati in corso d'opera. Ad esempio, avendo un array di 4 elementi, possiamo modificare il 6° elemento dell'array, accedendovi come se esistesse:

```

>> D = [1:2:7]
>> D(6) = 11

D = 1x4
     1     3     5     7

D = 1x6
     1     3     5     7     0    11

```

Alle posizioni non definite, come possiamo notare, viene assegnato il valore 0.

Un'operazione particolare, soprattutto quando si usano i vettori (vedi dopo) è l'elevamento a potenza. Normalmente l'elevamento a potenza viene fatto con la notazione `^n`, tuttavia esiste una seconda notazione: la notazione `.^n` indica che viene fatta la potenza alla

n di ogni singola componente di un vettore. Senza il `.` allora verrebbe fatto l'elevamento a potenza di un vettore moltiplicando il vettore per se stesso. Ad esempio:

```

Terminal
>> D = [3, 7; 14, 21];
>> D^2
>> D.^2

ans = 2x2
    107    168
    336    539

ans = 2x2
     9    49
    196   441

```

2.2.2 Celle

Un'ultima struttura dati che è molto comoda è la cell. Una cell è un array di valori dinamico, dove i valori possono avere tipi diversi. Una cell si definisce in questo modo:

```

Terminal
>> C = {"Hello world", [4, 6, 8], 16, -92.77, [16, 32, 11; 4, 97, 6; 14, 9, 8]}

```

	1	2	3	4	5
1	"Hello world"	[4,6,8]	16	-92.7700	[16,32,11;4,97,6;14,9,8]

Possiamo controllare il valore all'interno di un elemento di una cella con la notazione `cella(indice)`:

```

Terminal
>> C(1)

ans = 1x1 cell array
    {"Hello world"}

```

Possiamo anche estrarre un elemento da una cella tramite la notazione `cella{indice}`:

```

Terminal
>> A = C{1}

A = "Hello world"

```

Una cell non contiene i puntatori alle variabili inserite all'interno di essa, ma contiene delle copie identiche dei dati. Questo vuol dire che se una certa variabile viene inserita dentro la cell, se si modificasse la variabile allora il valore nella cell non si aggiornerebbe. I valori in una cell possono essere modificati usando la seguente notazione:

```
Terminal
>> C(1) = {'Ciao mondo'}
```

	1	2	3	4	5
1	"Ciao mondo"	[4,6,8]	16	-92.7700	[16,32,11;4,97,6;14,9,8]

2.2.3 Structs

Una struct in MATLAB è simile a una struct in C: questa ha dei campi nominali che possono essere modificati a piacimento. Ad esempio:

```
Terminal
>> S.string = "Hello world"
>> S.value = 32
>> S.matrix = [2, 4; 5, 7]
```

```
S = struct with fields:
    string: "Hello world"

S = struct with fields:
    string: "Hello world"
    value: 32

S = struct with fields:
    string: "Hello world"
    value: 32
    matrix: [2x2 double]
```

Possiamo estrarre i valori all'interno di un campo chiamando il campo stesso:

```
Terminal
>> S.value

ans = 32
```

SEZIONE 2.3 Funzioni

Come molti linguaggi di programmazione, MATLAB consente lo sviluppo di funzioni definite dall'utente, e questo si fa tramite la seguente notazione:

```
function [output1, output2, ...] = nome_funzione(input1, input2, ...)
    % Codice...
end
```

Segue un esempio di una funzione che calcola il Δ di un'equazione di secondo grado:

```
function [out] = custom_delta(a, b, c)
% Calcola il delta di una data equazione di secondo grado, dati i coefficienti
% a, b e c
    out = b^2 - 4*a*c;
end
```

Possiamo testarne il funzionamento chiamando la funzione stessa, considerando ad esempio la funzione $4x^2 + 11x - 3 = 0$:

```
>> [d1] = custom_delta(4, 11, -3)

d1 = 169
```

2.3.1 Funzioni particolari

Alcune funzioni particolari di MATLAB, che potrebbero risultare utili, sono le seguenti:

- `diff(x)`: dato un vettore x di n elementi, `diff(x)` calcola un vettore di $n - 1$ elementi dove ogni elemento y_i è uguale alla seguente operazione: $y_i = -x_i + x_{i+1}$. Tale funzione può anche avere un parametro extra, l'ordine. Questo parametro indica che `diff(x)` viene chiamato ricorsivamente tante volte quanto specifica l'ordine. Con le matrici, la differenza viene fatta considerando le righe della matrice. Partendo da una matrice A di dimensioni $m \times n$, la matrice finale avrà come dimensioni $(m - 1) \times n$. Ad esempio:

```
>> A = [-2, 5, -3, 6, -1, 4];
>> B = [4, 7, 8; 11, 4, -3; 16, 7, 9];

>> % Diff(A) con ordine 1
>> diff(A)

>> % Diff(A) con ordine 2
>> diff(A, 2)

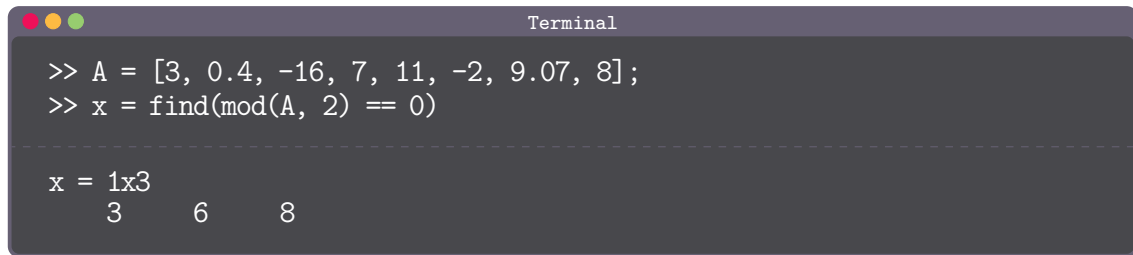
>> % Diff(B) con ordine 1
>> diff(B)

ans = 1x5
     7    -8     9    -7     5

ans = 1x4
    -15    17   -16    12

ans = 2x3
     7    -3   -11
     5     3    12
```


- `find(cond)`: data una condizione, `find()` ritorna tutti gli indici in un vettore che rispettano la condizione. Ad esempio:



```
Terminal
>> A = [3, 0.4, -16, 7, 11, -2, 9.07, 8];
>> x = find(mod(A, 2) == 0)

x = 1x3
     3     6     8
```

SEZIONE 2.4

Costrutti di flow control

Ci sono vari costrutti di flow control presenti in MATLAB, quali il for loop, il while loop e il condizionale if - else if - else.

CAPITOLO 3

Sistemi di Equazioni Non Lineari

Fino ad ora siamo sempre stati abituati a problemi analitici dove la soluzione a un problema era data da un'equazione, o al più un piccolo sistema di equazioni non lineari. Tuttavia nella realtà sono molti i casi dove la soluzione viene trovata risolvendo complessi sistemi di equazioni non lineari, che non sempre possono essere svolti a mano. Da qui, la nascita di alcuni metodi di calcolo numerico che aiutano nella risoluzione di tali sistemi. Prima di illustrare questi metodi, ci soffermeremo brevemente su cos'è un'equazione non lineare:

Equazione non lineare

DEFINITION

Un'**equazione non lineare** è un'equazione avente la forma

$$f(x) = 0$$

Chiamiamo **soluzione** ξ (o alternativamente **radici dell'equazione** o **zeri della funzione** f) di un'equazione non lineare quel valore tale che

$$f(\xi) = 0$$

All'interno di questo capitolo ci limiteremo prevalentemente al caso di radici reali. Per applicare un metodo su una funzione tuttavia, ci serve prima sapere le seguenti tre informazioni:

- 1) **quante** sono le radici (in questo caso, reali);
- 2) **dove** si trovano, approssimativamente, le radici;
- 3) se sono presenti delle **simmetrie** nella funzione.

Ci sono vari metodi per trovare queste informazioni: si può procedere allo **studio analitico**, alla **tabulazione** o all'analisi del **grafico** della funzione stessa. Procederemo ad illustrare tutti e tre i metodi su un'equazione di esempio:

3.0.1

EXAMPLE

Si consideri la seguente funzione $f(\lambda)$, che modella il tasso di crescita di una popolazione:

$$f(\lambda) = e^\lambda + \frac{0,435}{\lambda}(e^\lambda - 1) - 1,564 = 0$$

Procediamo a considerare lo **studio analitico** di questa funzione: notiamo che la funzione risulta definita e continua in $\mathbb{R}/\{0\}$, e studiando il semiasse positivo (non ha senso controllare il semiasse negativo, poiché quest'equazione modella la

crescita della popolazione) notiamo che:

$$\lim_{\lambda \rightarrow 0} f(\lambda) < 0 \quad \text{e} \quad \lim_{\lambda \rightarrow +\infty} f(\lambda) = +\infty$$

Calcolando la derivata prima, otteniamo invece la seguente funzione:

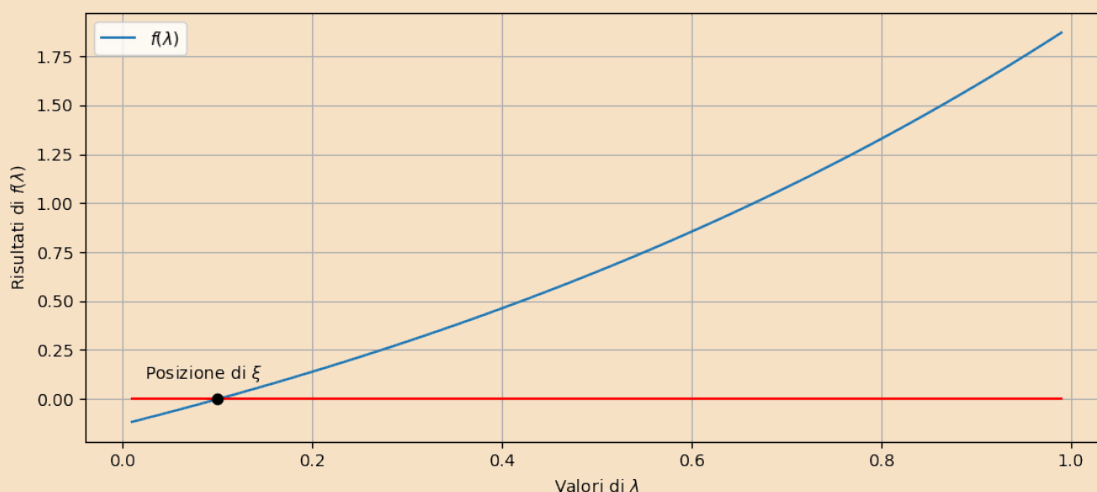
$$f'(\lambda) = e^\lambda + \left(1 + 0,435 \frac{\lambda - 1}{\lambda^2}\right) + \frac{0,435}{\lambda^2} > 0$$

Notiamo infatti che il comportamento della funzione è positivo oltre lo 0: questo significa che la funzione $f(\lambda)$ è monotona crescente. Possiamo dunque concludere che, nel semiasse positivo, sia presente un unico zero ξ .

Per il metodo della **tabulazione**, si considerano i valori ottenuti dalla funzione in corrispondenza di valori equidistanti di λ , e si osserva dunque il segno dei valori ottenuti. Ad esempio:

λ	$f(\lambda)$
0,10	-0,001335588295285
0,12	0,025672938554613
0,14	0,053195959592184
0,16	0,081243551500795
0,18	0,109825990666185
0,20	0,138953757158539

Come possiamo notare, abbiamo un cambio di segno tra $\lambda = 0,10$ e $\lambda = 0,12$: questo vuol dire che la radice della funzione si trova nell'intervallo $[0,10, 0,12]$. Possiamo anche osservare la radice usando un grafico della funzione:



Come abbiamo potuto notare dall'esempio, grazie ai tre metodi possiamo trovare una posizione approssimativa delle varie radici di una funzione. Ma come mai ci interessa tanto sapere la posizione di dove la funzione cambia segno? Perché grazie al **teorema di Bolzano**, questo ci permette di localizzare una radice.

Teorema di Bolzano

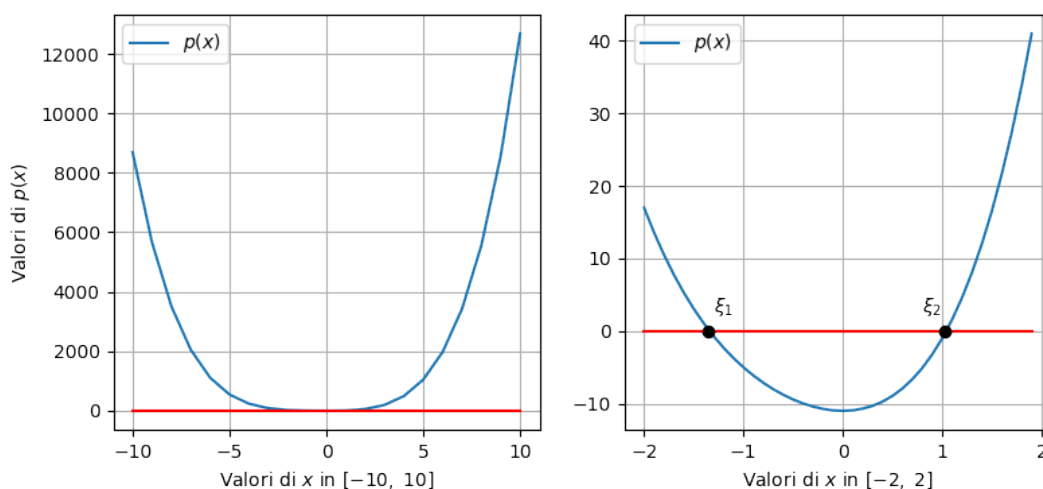
THEOREM

Dato un **intervallo** $[a, b]$ e una **funzione** $f(x)$ **continua**, se $f(a)$ ha **segno discorde** rispetto a $f(b)$ (quindi, se $f(a) \cdot f(b) < 0$), allora $f(x)$ **interseca almeno una volta** l'asse delle x

È importante tuttavia sapere anche restringere l'intervallo di osservazione delle radici. Supponiamo di avere in esame la funzione

$$p(x) = x^4 + 2x^3 + 7x^2 - 11 = 0$$

mostriamo qui due grafici della funzione, in intervalli diversi:



Notiamo come, in base all'intervallo, è più semplice notare la posizione delle radici. Infatti, $p(x)$ ha 4 radici, di cui due reali e due complesse coniugate.

SEZIONE
3.1

Metodo di bisezione (o dicotomico)

Tra i vari metodi utilizzabili per trovare le radici in una funzione, il più semplice e immediato da utilizzare è il **metodo di bisezione**, o **metodo dicotomico**. Questo metodo permette, una volta individuato un intervallo di separazione in cui si trova una **singola radice**, di costruire una successione $\{x_k\}$ di approssimazioni di ξ . Per applicare dunque questo metodo vanno rispettate due condizioni, dette **ipotesi di applicabilità**:

- è stato individuato un intervallo $I = [a, b]$, all'interno del quale è presente **un'unica radice** ξ ;
- la funzione f in esame deve essere **continua in I** (formalmente, $f \in C^0[a, b]$, dove C^0 è l'insieme di funzioni continue);
- i due estremi a e b devono avere **segno discorde** (dunque $f(a) \cdot f(b) < 0$).

In sintesi, il teorema di Bolzano deve essere rispettato all'interno del nostro intervallo I ; il metodo di bisezione infatti usa estensivamente il suddetto teorema. Passiamo dunque ad esaminare l'algoritmo del metodo di bisezione:

Algorithm 1: Metodo di bisezione (o dicotomico)**Input:** L'intervallo $[a, b]$, la funzione $f(x)$ e la tolleranza

```

1  $a \leftarrow a_0, b \leftarrow b_0;$ 
2  $\xi_{\text{seq}} \leftarrow \{ \};$  //  $\xi_{\text{seq}}$  è una sequenza vuota
3 for  $k$  in 1, 2, 3, ... do
4    $x_k \leftarrow \frac{a+b}{2};$ 
5    $d \leftarrow |x_k - x_{k-1}|;$ 
6   Aggiungere  $x_k$  a  $\xi_{\text{seq}};$ 
7   // Se si trova la radice oppure viene raggiunta la tolleranza, l'algoritmo si ferma
8   if  $(f(x_k) = 0)$  or  $(d < \text{tol})$  then
9     return  $\xi_{\text{seq}}$ 
10  end
11  // In base all'intervallo che contiene la radice, ripetere l'algoritmo con l'intervallo aggiornato
12  if  $f(a) \cdot f(x_k) < 0$  then
13     $a \leftarrow a, b \leftarrow x_k$ 
14  end
15  if  $f(x_k) \cdot f(b) < 0$  then
16     $a \leftarrow x_k, b \leftarrow b$ 
17  end
18 end

```

Per il metodo di bisezione, l'idea è che dato un intervallo $I = [a, b]$, **dividendo** I sempre **in sotto-intervalli** più contenuti, riusciremmo eventualmente ad ottenere un intervallo più piccolo all'interno del quale troveremmo la nostra radice ξ . Ogni sotto-intervallo è costituito da una delle due metà di I . Per sapere quale sotto-intervallo contiene ξ , basta applicare il teorema di Bolzano. L'algoritmo è semplice, e genera una successione di tutte le approssimazioni di ξ , denominata $\{x_k\}$ (o, nell'algoritmo, ξ_{seq}). La precisione del metodo di bisezione è ottenibile calcolando il relativo **errore di troncamento**.

Errore di troncamento

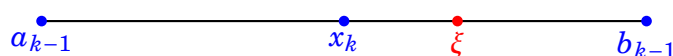
L'**errore di troncamento** è l'errore commesso **approssimando** la radice ξ con il k -esimo elemento della successione creata tramite l'algoritmo del metodo di bisezione

$$e_k = \xi - x_k$$

Ma l'algoritmo **può convergere**? Intuitivamente, convergerà verso ξ solo se l'errore si dovesse ridurre a 0. Formalmente, possiamo esprimere questa relazione come

$$\lim_{k \rightarrow \infty} x_k = \xi \iff \lim_{k \rightarrow \infty} |e_k| = 0$$

Possiamo però esprimere e_k anche in altri termini. Per il metodo di bisezione, noi sappiamo che alla k -esima iterazione, ξ sarà presente solo in $[a_{k-1}, x_k]$ o in $[x_k, b_{k-1}]$.



Dunque, data una generica iterazione x_k , l'errore di troncamento alla suddetta sarà uguale a

$$|e_k| < \frac{b_{k-1} - a_{k-1}}{2}$$

Ora, siccome l'intervallo $[a_{k-1}, b_{k-1}]$ ha ampiezza pari alla metà dell'intervallo all'iterazione precedente (dunque $[a_{k-2}, b_{k-2}]$), possiamo costruire anche una formula generica dell'errore per qualsiasi iterazione k :

$$|e_k| < \frac{b_{k-1} - a_{k-1}}{2} = \frac{b_{k-2} - a_{k-2}}{2^2} = \dots = \frac{b - a}{2^k}$$

Dunque, anche il limite di prima può essere riscritto come

$$0 \leq \lim_{k \rightarrow \infty} |e_k| < \lim_{k \rightarrow \infty} \frac{b - a}{2^k} = 0$$

3.1.1 Ordine di convergenza e criteri di arresto

Abbiamo visto che il metodo di bisezione converge, ma è anche importante che converga in tempi rapidi. Come possiamo determinare la "velocità" di convergenza? Questo viene determinato in base a un valore chiamato **ordine di convergenza** p .

Ordine e Fattore di Convergenza

DEFINITION

Sia $\{x_k\}$ una successione di approssimazioni **convergente** a ξ . Si dice che la successione ha un **ordine di convergenza** p e un **fattore di convergenza** C se esistono due numeri reali $p \geq 1$ e $C > 0$ tali che

$$\lim_{k \rightarrow \infty} \frac{|e_{k+1}|}{|e_k|^p} = C$$

Se $p = 1$, si dice che la convergenza è **lineare**, se $p = 2$, si dice invece che la convergenza è **quadratica**.

Applicando la definizione di ordine e fattore di convergenza al metodo di bisezione, otteniamo che, per $k \rightarrow \infty$, si ha:

$$\frac{|e_{k+1}|}{|e_k|^p} = \frac{\frac{b-a}{2^{k+1}}}{\frac{b-a}{2^k}} = \frac{2^k}{2^{k+1}} = \frac{2^k}{2^k} \cdot \frac{1}{2} = \frac{1}{2}$$

Cosa ci dice il risultato appena ottenuto? Che, supponendo una convergenza lineare, otteniamo un fattore di convergenza di $1/2$. Questo ci dice che la convergenza è in realtà **lenta**: ad ogni step dell'algoritmo riusciamo a dimezzare l'errore, e guadagniamo una cifra binaria per meglio esprimere il nostro risultato. Siccome $2^{-4} < 10^{-1} < 2^{-3}$, allora ogni 3 o 4 iterazioni si riesce a guadagnare una cifra decimale.

Tuttavia, a causa degli errori di arrotondamento e troncamento da parte del computer, è praticamente impossibile che si riesca a raggiungere $f(x_k) = 0$. Dunque, quando dovremmo interrompere i calcoli? Possiamo definire dei **criteri di arresto a posteriori**, ovvero

$$\begin{cases} |e_k| \simeq |x_k - x_{k-1}| < \epsilon & \text{Se l'errore diventa minore di una tolleranza } \epsilon \dots \\ |f(x_k)| < \epsilon & \dots \text{o se la funzione ritorna numeri minori della tolleranza } \epsilon \end{cases}$$

Nel caso dell'algoritmo di bisezione, è stato scelto in precedenza di usare il primo criterio, ma potevano essere usati entrambi i criteri. Possiamo anche calcolare **a priori** una stima

di quante iterazioni K avremo bisogno prima di ottenere un errore minore di ϵ_{\min} . Per farlo, ci avvaliamo della formula dell'errore di troncamento:

$$|e_k| < \frac{b-a}{2^k} < \epsilon_{\min} \implies K > \frac{\log(b-a) - \log(\epsilon_{\min})}{\log(2)}$$

K dovrà essere arrotondato all'intero più vicino, in quanto deve essere un intero positivo.

SEZIONE 3.2

Metodo di Newton-Raphson

Forse uno dei metodi più utilizzati quando si parla di equazioni non lineari, il **metodo di Newton-Raphson** (o metodo delle tangenti) è un metodo molto usato, grazie alla sua convergenza più immediata rispetto al metodo di bisezione. Questo metodo prevede di approssimare una funzione $f(x)$ in un intorno I , all'interno del quale si deve trovare una e una sola radice ξ , grazie alle sue **tangenti** in vari punti, le quali vengono calcolate grazie ad uno **sviluppo in serie di Taylor**.

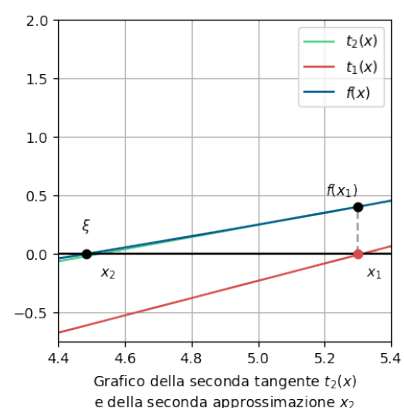
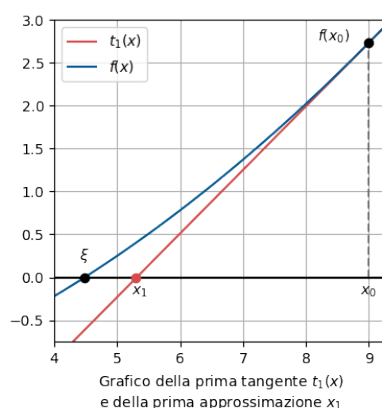
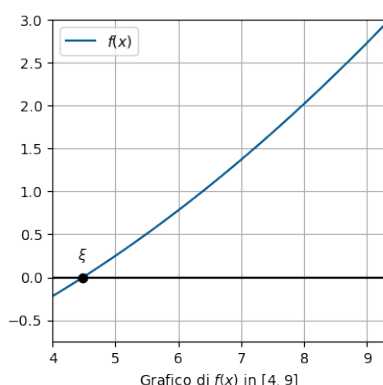
Partendo da un'approssimazione iniziale della radice, che chiameremo x_0 , è possibile costruire la tangente t_0 alla funzione nel punto di intersezione tra $x = x_0$ e la funzione stessa (quindi, nel punto $(x_0, f(x_0))$). Per costruire la tangente, viene utilizzato lo sviluppo in serie di Taylor (**fino al primo ordine**), che tramite il calcolo della derivata prima permette di trovare l'equazione di t_0 :

$$t_0 = f(x_0) + f'(x_0) \cdot (x - x_0)$$

Calcolata la tangente, si procede recuperando il punto di intersezione tra t_0 e l'asse delle x ; tale punto viene chiamato x_1 . Una volta ottenuto x_1 , il procedimento ricomincia da capo: si trova dunque il punto di intersezione $(x_1, f(x_1))$, si calcola la tangente t_1 nel punto appena trovato (quindi $t_1 = f(x_1) + f'(x_1) \cdot (x - x_1)$) e si ottiene un nuovo punto, chiamato x_2 . Il procedimento continua finché non si raggiunge o la radice o un criterio d'arresto.

Generalmente, ad ogni iterazione $k = 1, 2, \dots$ del metodo, la nuova approssimazione della radice x_k è data dall'intersezione tra la tangente t_k a $f(x)$ nel punto $(x_{k-1}, f(x_{k-1}))$ e l'asse delle x :

$$t_k = f(x_k) + f'(x_k) \cdot (x - x_k) \implies \text{si vuole risolvere } f(x_k) + f'(x_k) \cdot (x - x_k) = 0$$



Possiamo tuttavia rifattorizzare l'espressione, così da renderla in funzione dell'approssimazione in un'iterazione k :

$$\begin{aligned}
 f(x_{k-1}) + f'(x_{k-1}) \cdot (\underbrace{x}_{\text{becomes } x_k} - x_{k-1}) &= 0 \\
 f(x_{k-1}) + x_k \cdot f'(x_{k-1}) - x_{k-1} \cdot f'(x_{k-1}) &= 0 \\
 -x_k \cdot f'(x_{k-1}) &= f(x_{k-1}) - x_{k-1} \cdot f'(x_{k-1}) \\
 -x_k &= \frac{f(x_{k-1})}{f'(x_{k-1})} - x_{k-1} \cdot \frac{f'(x_{k-1})}{f'(x_{k-1})} \\
 x_k &= x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}
 \end{aligned}$$

L'algoritmo, dal punto di vista matematico, diventa dunque il seguente:

$$\begin{cases} x_0 & \text{viene dato come input} \\ x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})} & \text{per } k = 1, 2, \dots \end{cases}$$

Viene qui mostrato l'algoritmo del metodo:

Algorithm 2: Metodo di Newton-Raphson (o delle tangenti)

Input: La funzione $f(x)$, un'approssimazione iniziale x_0 e la tolleranza

```

1  $x_k \leftarrow x_0$ ;
2  $\xi_{\text{seq}} \leftarrow \{\}$ ;

// Finché  $x_k$  non è uguale a  $\xi$  oppure finché non si raggiunge la tolleranza...
3 while ( $f(x_k) \neq 0$ ) or ( $x_k > \text{tol}$ ) do
    // ...calcola la nuova approssimazione di  $\xi$ 
4      $x_{k+1} \leftarrow x_k - \frac{f(x_k)}{f'(x_k)}$ ;
5     Aggiungere  $x_{k+1}$  a  $\xi_{\text{seq}}$ ;
6      $x_k \leftarrow x_{k+1}$ ;
7 end
8 return  $x_k, \xi_{\text{seq}}$ 

```

Procediamo a mostrare questo metodo attraverso un esempio:

EXAMPLE

💡

3.2.1

Calcolare la radice quadrata di un numero a è equivalente al trovare gli zeri della funzione

$$f(x) = x^2 - a$$

Per trovare gli zeri, vogliamo ricorrere al metodo di Newton-Raphson. Modellando l'algoritmo definito in precedenza, otteniamo la seguente equazione:

$$\begin{cases} x_0 & \text{dato come input} \\ x_k = x_{k-1} - \frac{x_{k-1}^2 - a}{2x_{k-1}} & \text{per } k = 1, 2, \dots \end{cases}$$

Possiamo ulteriormente semplificare l'algoritmo, raggiungendo dunque la

EXAMPLE

seguinte equazione:

$$\begin{cases} x_0 & \text{dato come input} \\ x_k = \frac{1}{2} \cdot (x_{k-1}) & \text{per } k = 1, 2, \dots \end{cases}$$

Similmente al metodo di bisezione, possiamo definire se l'algoritmo di Newton-Raphson convergerà, definendo dunque le sue condizioni di convergenza.

Convergenza del metodo di Newton-Raphson: esistenza di J

THEOREM

Data una funzione $f(x)$, se:

- è stato separato un intervallo $I = [a, b]$ dove c'è una **singola** radice ξ ;
- f, f', f'' sono continue in I , tale che $f \in C^2[a, b]$;
- $f'(x) \neq 0$ per $x \in [a, b]$;

allora esiste un intorno $J \subseteq I$ di ξ tale che, se $x_0 \in J$, allora la successione delle approssimazioni convergerà a ξ .



Il precedente teorema non stabilisce che il metodo convergerà, ma che piuttosto esiste un intervallo J , ristretto rispetto a I , nel quale troveremo la nostra radice ξ .