Introduzione alla programmazione per le scuole superiori

basics

Nov 07, 2024

CONTENTS

I Introduzione alla programmazione **3**

1 Introduzione alla programmazione - in Python **5**

2 Variabili, tipi e funzioni elementari built-in **7**
2.1 Commenti 7
2.2 Tipi built-in 7

3 Controllo del flusso **11**
3.1 Alternativa 11
3.2 Iterazione 12

4 Funzioni **15**
4.1 Default value of function arguments 15

5 Classi **17**

6 Librerie **19**

II Introduzione al calcolo scientifico **21**

7 Introduzione al calcolo scientifico **23**
7.1 Sistemi lineari 24
7.2 Equazioni algebriche non lineari 27
7.3 Approssimazione di funzioni 33
7.4 Derivate di funzioni 33
7.5 Integrali 35
7.6 Equazioni differenziali ordinarie 39
7.7 Ottimizzazione 40

III Introduzione ai metodi in statistica e AI **41**

8 Introduzione ai metodi in AI **43**

IV Supporto tecnico **45**

9 Supporto tecnico **47**

Questo libro fa parte del materiale pensato per le scuole superiori. E' disponibile la versione in .pdf scaricabile.

Obiettivi generali. Questo lavoro punta ad essere un'opera di *formazione alla* **tirchieria**, **prigrizia** *e* **onestà**, almeno in ambito informatico. I più benvolenti potranno riassumere questo obiettivo come il desiderio di *non buttare nel WC soldi, tempo, e pazienza*.

Oltre alle nozioni minime, l'obiettivo principale di questo libro è la formazione a:

- **indipendenza** in ambito informatico: evitare di pagare per qualcosa che non serve; evitare di pagare per qualcosa di inutile o dannoso in presenza di alternative libere; meglio dedicare le risorse a ciò che vale la pena pagare
- **ordine**: i moderni strumenti informatici permettono di lavorare in maniera ordinata, risparmiando tempo, soldi e pazienza
- **trasparenza** e **onestà**: anche se spesso in maniera non lineare, la conoscenza procede seguendo il metodo scientifico: i risultati mostrati e le tesi proposte devono essere supportate da dati e logica; i dati e le analisi svolte per poter produrre risultati devono essere disponibili, controllabili e soggetti a critica. Tutto il resto, almeno qui, almeno nell'ambito della conoscenza che procede con il metodo scientifico, sarà considerata confusione nella migliore delle ipotesi o direttamente *merda*.

Questo stesso libro è scritto seguendo questi criteri: oltre al dispositivo elettronico usato per consultare il materiale (online o offline, una volta scaricato), non è necessaria la spesa per nessun altro dispositivo o infrastruttura informatica; i sorgenti del materiale è sviluppato localmente, ospitato e disponibile su Github all'indirizzo https://github.com/Basics2022/bbooks-programming-hs.

necessità di una connessione internet, se non si porta il progetto su un sistema locale, con tutti gli strumenti necessari - non tanti, e standard, ma comunque devono esserci "per funzionare"

Metodo.

- **Impostazione degli strumenti necessari. todo**…
- **Linguaggio di programmazione.** In questa introduzione si sceglie di usare **Python** come linguaggio di programmazione. Un approccio più generale all'informatica e alla programmazione prevederebbe l'utilizzo di altri linguaggio di programmazione (come C). Considerata la **diffusione** di Python, la quantità di **librerie disponibili** (con eventuali binding a librerie sviluppate usando linguaggi di programmazione più efficienti) e strumenti per la **programmazione collaborativa** e remota (**Colab**, **Jupyter**,…), Python risulta comunque una buona scelta per un corso improntato alla presentazione delle basi di programmazione dirette a un'applicazione abbastanza immediata.

  Verranno posta attenzione sulla sintassi "particolare" di Python quando si discosta maggiormente dagli altri linguaggi di programmazione.
- **Argomenti. todo**…

Part I

Introduzione alla programmazione

CHAPTER

ONE

INTRODUZIONE ALLA PROGRAMMAZIONE - IN PYTHON

L'introduzione alla programmazione in Python userà le risorse messe a disposizione da Google con il progetto Jupyter per lo sviluppo di codice open-source, con open-standard e servizi interattivi utilizzabili su diversi dispositivi usando diversi linguaggi di programmazione, come Python, Julia o R

Oltre al dispositivo elettronico utilizzato per consultare al materiale, non è necessario nessun altro dispositivo informatico: un account Google personale permette l'accesso libero ai servizi base di cloud computing di Colab

CHAPTER

TWO

VARIABILI, TIPI E FUNZIONI ELEMENTARI BUILT-IN

- tipi
- variabili
  - …
  - by-reference o by-value

2.1 Commenti

```
# I commenti permettono di aggiungere brevi descrizione al codice

# In Python, è possibile aggiungerre commenti al codice con il carattere #: tutto␣
↪quello
# che viene dopo il carattere # su una riga è considerato un commento, e non codice da
# eseguire

# E' buona regola aggiungere qualche commento al codice, e sarebbe bene iniziare a␣
↪farlo
# in lingua inglese:
# - il codice non è auto-esplicativo!
# - il codice potrebbe essere usato da altri in giro per il mondo, ed è più probabile␣
↪che
#   si conosca l'inglese invece dell'italiano

# So let's switch to English for scripts, both for comments and "for variable names"

# Comments are not documentation! **todo** Add some paragraph about documentation!
```

2.2 Tipi built-in

- numero: intero, reale, complesso
- booleano
- stringa
- bytes
- lista
- tupla

- insieme
- dizionario, `dict`

2.2.1 Numeri

```
"""
Numbers

in Python, variables are not delcared. Thus, a number variable is not defined
as an integer, a real or a complex variable, but it's type is inferred by its
initialization
"""

# Numbers
a_int = 1
a_real = 1.
a_complex = 1.+ 0.j

# Strings
a_str = '1.0'

print(f"type(a_int)    : {type(a_int    )}")
print(f"type(a_real)   : {type(a_real   )}")
print(f"type(a_complex): {type(a_complex)}")
print(f"type(a_str    ): {type(a_str    )}")
```

```
type(a_int)    : <class 'int'>
type(a_real)   : <class 'float'>
type(a_complex): <class 'complex'>
type(a_str    ): <class 'str'>
```

2.2.2 Booleani - logici

2.2.3 Stringhe

```
"""
Strings

strings are character
"""
```

```
'\nStrings\n\nstrings are character \n'
```

2.2.4 Liste, tuple e insiemi

2.2.5 Dizionari

CHAPTER

THREE

CONTROLLO DEL FLUSSO

Nei paradigmi di *programmazione imperativa* (**todo** fare riferimento ai paradigmi di programmazione. Ha senso questa distinzione?), vengono usate delle strutture di controllo del flusso di esecuzione di un programma.

Si possono distinguere due categorie delle strutture di controllo:

- condizionale ed alternativa: if, if-then, if-then-else
- iterazione: for, while, …

3.1 Alternativa

3.1.1 `if-then` statement

```
""" if-then example """
# Try this script changing the user input

# User input
a = 2         # a is initialize as an integer
# a = 2.1       # if a is initialized as a real, a % 2 perform automatic casting,␣
↪int(a) % 2
                # uncomment previous line and try!

word = 'odd'
if ( a % 2 == 0 ):    # automatic casting into an int -> a % 2 = floor(a) % 2
    word = 'even'

print(f"User input ({a}) is an {word} number")
```

```
User input (2) is an even number
```

3.1.2 `if-then-else` statement

```
""" if-then-else example"""
# Try this script changing the user input

# User input
a = 15

if ( a % 3 == 1 ):      # First condition
    reminder = 1
elif ( a % 3 == 2 ):    # Other condition
    reminder = 2
else:                   # All the other conditions
    reminder = 0

print(f"User input ({a}). {a} % 3 = {reminder}")
```

```
User input (15). 15 % 3 = 0
```

3.2 Iterazione

3.2.1 for loop

```
""" for loop examples

Loops over:
- elements of a list
- elements in range
- keys, values of a dict
- ...
"""

# Loop over elements of a list
seq = ['a', 3, 4. , {'key': 'value'}]

print("\nLoop over elements of the list: seq = {seq}")
for el in seq:
    print(f"element {el} has type {type(el)}")

# Loop over elements of a tuple
# ...

# Loop over elements of a range
n_el = 5
range_el = range(5)
print("\nLoop over elements of the list, seq = {seq}")
print(f"range({n_el}) has type: {type(range_el)}")
print(f"range({n_el}): {range_el}")

for i in range_el:
    print(i)
```

(continues on next page)

(continued from previous page)

```
# Loop over keys, values of a dict
d = {'a': 1., 'b': 6, 'c': {'c1': 1, 'c2': True}}
print(f"\nLoop over elements of the dict, d = {d}")

for i,k in d.items():
    print(i, k)
```

```
Loop over elements of the list: seq = {seq}
element a has type <class 'str'>
element 3 has type <class 'int'>
element 4.0 has type <class 'float'>
element {'key': 'value'} has type <class 'dict'>

Loop over elements of the list, seq = {seq}
range(5) has type: <class 'range'>
range(5): range(0, 5)
0
1
2
3
4

Loop over elements of the dict, d = {'a': 1.0, 'b': 6, 'c': {'c1': 1, 'c2': True}}
a 1.0
b 6
c {'c1': 1, 'c2': True}
```

3.2.2 while loop

```
""" while loop example """

a = 3

while ( a < 5 ):
    a += 1
    print(f">> in while loop, a: {a}")

print(f"after while loop, a: {a}")
```

```
>> in while loop, a: 4
>> in while loop, a: 5
after while loop, a: 5
```

3.2.3 altri cicli

todo

CHAPTER

FOUR

FUNZIONI

```
#> Define a function to tell if a number is positive or not
def is_positive(x):
    """ Function returning True if x>0, False if x<=0 """
    return x > 0

#> User input to test the function: tuple of numbers
n_tuple = [ -1, 2., .003, 1./3., -2.2, 0, -7.4 ]

#> Test function on all the elements in the user-defined tuple
for n in n_tuple:
    if ( is_positive(n) ):
        string = ''
    else:
        string = 'not '

    print(f'{n} is '+string+'positive')
```

```
-1 is not positive
2.0 is positive
0.003 is positive
0.3333333333333333 is positive
-2.2 is not positive
0 is not positive
-7.4 is not positive
```

4.1 Default value of function arguments

```
#> Define a user function to tell is the first argument is greater than the second.
# If no second argument is given, it's set = 0 (default value, defined in the␣
 ↪function)
def is_greater_than(x, y=0):
    """  """
    return is_positive(x-y)

a, b = -2, -3
print(f"Is {a} greater than {b}? is_greater_than({a}, {b}):"+str(is_greater_than(a,␣
 ↪b)))
print(f"Is {a} greater than  0 ? is_greater_than({a},    ):"+str(is_greater_than(a)))
```

```
Is -2 greater than -3? is_greater_than(-2, -3):True
Is -2 greater than  0 ? is_greater_than(-2,    ):False
```

CHAPTER

FIVE

CLASSI

- classi, metodi, oggetti
- ereditarietà, overloading

CHAPTER

SIX

LIBRERIE

- Definizione del concetto
- librerie “standard”, scritte da qualcun’altro
- scrivere una libreria

Part II

Introduzione al calcolo scientifico

CHAPTER

SEVEN

INTRODUZIONE AL CALCOLO SCIENTIFICO

In questa introduzione al calcolo numerico, vengono presentati alcuni algoritmi. Dove sensato, viene implementata la versione elementare di alcuni di questi algoritmi. Per usi non didattici, e quando possibile, si raccomanda l'uso di algoritmi implementati in librerie disponibili, per questioni di tempo ed efficienza: è lavoro già fatto, da persone che lo sanno fare meglio di noi, controllato, migliorato nel corso degli anni, ottimizzato per ogni sistema e spesso in linguaggi di programmazione diversi da Python, come C o Fortran.

In questa introduzione viene fatto affidamento e uso di alcune librerie disponibili per Python:

- librerie con algoritmi e strumenti matematici per il calcolo numerico: NumPy, SciPy,…
- librerie per la creazione di grafici: Matplotlib, Plotly,…
- librerie per l'analisi dati e la statistica: pandas,…
- librerie per il machine learning: sci-kit, PyTorch,…
- …

Introduzione al calcolo numerico

- Equazioni lineari
- Equazioni non lineari
- Approssimazione di funzioni
- Derivate
- Integrali
- Equazioni differenziali ordinarie:
  - problema di Cauchy ai valori iniziali
  - problema ai valori al contorno
- Ottimizzazione, vincolata e non

Metodi per la statistica

Introduzione al machine learning

7.1 Sistemi lineari

La soluzione di sistemi lineari è un problema che compare in molte altre applicazioni di calcolo numerico.

Formalismo matriciale. Con il formalismo matriciale, un sistema di equazioni lineari può essere scritto come

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Classificazione. In generale, i sistemi di equazioni lineari possono essere classificati:

- in base al numero di incognite n_u ed equazioni indipendenti n_e: $n_e = n_u$ sistemi determinati con un'unica soluzione; $n_e > n_u$ sistemi sovradeterminati: con nessuna soluzione in generale; $n_e < n_u$ sistemi indeterminati, con infinite soluzioni in generale
- in base alla "struttura" del sistema:
  - diagonale, tridiagonale, …
- in base al numero di coefficienti non-nulli della matrice **A**: sistemi con matrice **A** piena o **sparsa**; questa distinzione non è netta, ma il più delle volte risulta chiara dalla particolare applicazione/metodo.

Algoritmi. Esistono due grandi classi di metodi/algoritmi per la soluzione di sistemi lineari:

- i **metodi diretti**, che si basano su una fattorizzazione della matrice
- i **metodi indiretti**, che si basano sul calcolo di prodotti matrice-vettore

7.1.1 Sistemi lineari quadrati con matrici piene

In questa sezione si discute la soluzione di sistemi lineari quadrati con matrici piene con le funzoni disponibili nella libreria NumPy.

Esempio 1. Sistema quadrato determinato

Il sistema lineare

$$\begin{cases} x_1 + 2\,x_2 = 0 \\ x_1 + x_3 = -1 \\ x_1 + x_2 + x_3 = 1 \end{cases}$$

può essere riscritto con il formalismo matriciale nella forma $\mathbf{A}\mathbf{x} = \mathbf{b}$,

$$\underbrace{\begin{bmatrix} 1 & 2 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}}_{\mathbf{b}}$$

e risolto grazie alla funzione `solve(A, b)` della libreria `numpy.linalg`.

```
"""
Linear systems with full square non-singular matrices
"""

import numpy as np

A = np.array([[1.,2.,0.], [1.,0.,1.], [1.,1.,1.]])
b = np.array([0.,-1.,1.])

x = np.linalg.solve(A, b)

print(f"Sol, x: {x}")
print(f"Proof : Ax = {A @ x}")    # Check that Ax = b
print(f"         b = {b}")
```

```
Sol, x: [-4.  2.  3.]
Proof : Ax = [ 0. -1.  1.]
         b = [ 0. -1.  1.]
```

Esempio 2. Sistemi quadrati non determinati

I sistemi lineari

$$\begin{cases} x_1 + 2\,x_2 = 1 \\ x_1 + x_3 = -1 \\ 2x_1 + 2\,x_2 + x_3 = 1 \end{cases} \quad , \quad \begin{cases} x_1 + 2\,x_2 = 0 \\ x_1 + x_3 = -1 \\ 2x_1 + 2\,x_2 + x_3 = 1 \end{cases}$$

sono due sistemi quadrati non determinati. Il primo sistema non ha soluzioni, mentre il secondo ne ha infinite della forma

$$(x_1, x_2, x_3) = (-2, 1, 1) + \alpha(2, -1, -2)\,, \quad \alpha \in \mathbb{R}\,.$$

Dopo aver riscritto i sistemi lineari con il formalismo matriciale, si può provare a risolverli usando la funzione `solve(A, b)` della libreria `numpy.linalg`. In entrambi i casi, la funzione `solve(A,b)` resitiuisce un errore, segnalando che la matrice del sistema lineare è singolare, definizione equivalente di sistemi non determinati.

- **todo** *dare interpretazione geometrica, fare grafico?*
- **todo** *spiegare motivo?*
  - *Esistono algoritmi che trovano almeno una soluzione nel caso in cui ne esistano infinite?*: discutere gli algoritmi implementati nella funzione $numpy.linalg.solve()$ e rimandare alla documentazione della libreria; discutere altri algoritmi che rendono possibile trovare una soluzione
  - *Esistono algoritmi che trovano una soluzione approssimata nel caso in cui non ne esistano?*: **minimi quadrati**, minimizzano l'errore, dare un'interpretazione geometrica

```
"""
Linear systems with full square singular matrices
"""

import numpy as np

A = np.array([[1.,2.,0.], [1.,0.,1.], [2.,2.,1.]])
b = np.array([1.,-1.,1.])
# b = np.array([0.,-1.,1.])
```

(continues on next page)

(continued from previous page)

```
x = np.linalg.solve(A, b)

print(f"Sol, x: {x}")
print(f"Proof : Ax = {A @ x}")    # Check that Ax = b
print(f"          b = {b}")
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
Cell In[2], line 11
      8 b = np.array([1.,-1.,1.])
      9 # b = np.array([0.,-1.,1.])
---> 11 x = np.linalg.solve(A, b)
     13 print(f"Sol, x: {x}")
     14 print(f"Proof : Ax = {A @ x}")    # Check that Ax = b

File <__array_function__ internals>:200, in solve(*args, **kwargs)

File ~/.local/lib/python3.8/site-packages/numpy/linalg/linalg.py:386, in solve(a,␣
↪b)
    384 signature = 'DD->D' if isComplexType(t) else 'dd->d'
    385 extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 386 r = gufunc(a, b, signature=signature, extobj=extobj)
    388 return wrap(r.astype(result_t, copy=False))

File ~/.local/lib/python3.8/site-packages/numpy/linalg/linalg.py:89, in _raise_
↪linalgerror_singular(err, flag)
     88 def _raise_linalgerror_singular(err, flag):
---> 89     raise LinAlgError("Singular matrix")

LinAlgError: Singular matrix
```

7.1.2 Matrici sparse

Una matrice sparsa ha un elevato numero di elementi nulli. Una matrice sparsa viene definita in maniera efficiente salvando in memoria solo gli elementi non nulli (**limiti di memoria**); gli algoritmi per le matrici sparse risultano spesso efficienti perché evitano un molte operazioni che darebbero risultati parziali nulli (**velocità**).

- **todo** dire due parole sui formati
- **todo** fare esempio di calcolo del prodotto matrice vettore per matrici sparse

Esempio 1 - Matrice di rigidezza di elementi finiti

Il sistema lineare

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix},$$

è descritto da una matrice, $N = 5$, $N \times N = 25$, che ha $N + (N-1) + (N-1) = 13$ elementi non nulli. Il rapporto tra il numero di elementi non nulli e il numero di elementi totali è $\frac{3N-2}{N^2} \sim \frac{3}{N}$. Al crescere della dimensione

del problema, la matrice **A** diventa sempre più sparsa e diventa sempre più conveniente definirla come matrice sparsa, ed usare gli algoritmi pensati per questo tipo di matrici.

```
"""
Linear systems with square non-singular matrices, in sparse format
"""

from scipy import sparse

# Printout level: the higher the number, the more verbose the script
printout_level = 1

n_nodes = 5
i_nodes = list(np.arange(5))

# Build sparse stiffness matrix, I: row indices, J: col indices, E: matrix elems
I = np.array(i_nodes+i_nodes[:-1]+i_nodes[ 1:])
J = np.array(i_nodes+i_nodes[ 1:]+i_nodes[:-1])
E = np.array(n_nodes*[2]+(n_nodes-1)*[-1]+(n_nodes-1)*[-1])

A = sparse.coo_array((E, (I,J))).tocsr()

if ( printout_level > 50 ):  # print matrix in sparse format
    print(f" I: {I}\n J: {J}\n E: {E}")
    print(f" A:\n {A}")

if ( printout_level > 60 ):  # convert and primt matrix in full format
    print(f" A.todense(): {A.todense()}")

# RHS
b = np.array(5*[1])

# Solve linear system
x = sparse.linalg.spsolve(A, b)

print(f"Sol, x: {x}")
```

```
Sol, x: [2.5 4.  4.5 4.  2.5]
```

7.2 Equazioni algebriche non lineari

Questa sezione si occupa della soluzione delle equazioni algebriche non lineari, distinguendo le equazioni non lineari con una sola incognita $x \in \mathbb{R}$

$$f(x) = 0\,,$$

e i sistemi di equazioni non lineari con un numero di incognite pari al numero di equazione,

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}\,.$$

7.2.1 Equazioni non lineari

Vengono presentati i metodi di bisezione e di Newton per la soluzione di un'equazione non lineare,

$$f(x) = 0\,,$$

e applicati alla soluzione del problema con $f(x) = e^x + x$, la cui derivata è nota e immediata da calcolare $f'(x) = e^x + 1$. L'espressione della derivata verrà utilizzata nel metodo di Newton.

```
""" Import libaries """

import numpy as np
from time import time
```

```
"""
Example. f(x) = e^x + x
"""

# Function f and its derivative
f  = lambda x: np.exp(x) + x
df = lambda x: np.exp(x) + 1
```

Metodo di bisezione

Il metodo di bisezione per la ricerca degli zeri di una funzione continua $F(x)$ si basa sul teorema dei valori intermedi per le funzioni continue.

Dati due numeri reali a, b tali che $f(a)\,f(b) < 0$, allora esiste un punto $c \in (a, b)$ tale che $f(c) = 0$.

```
"""
Define bisection_method_scalar() function to solve nonlinear scalar equations with␣
↪bisection method
"""

def bisection_method_scalar(f, a, b, tol=1e-6, max_niter=100):
    """ Function implementing the bisection method for scalar equations """

    niter = 0

    if ( not f(a) * f(b) < 0 ):
        print("Bisection algorithm can't start, f(a)f(b)>= 0")
    else:
        x = .5 * (a+b)
        fx = f(x)
        while ( np.abs(fx) > tol and niter < max_niter ):

            if ( f(x) * f(a) <= 0 ):  # new range [a,c]
                b = x
            else:  # new range [a,b]
                a = x

            # Update solution and residual
            x = .5 * (a+b)
            fx = f(x)
```

(continues on next page)

(continued from previous page)

```
            # Update n.iter
            niter += 1

    return x, np.abs(fx), niter, max_niter
```

```
""" Use bisection_method_scalar() function to solve the example """

# Find 2 values so that $f(a) f(b) < 0$
a, b = -2., 0.

t1 = time()
x, res, niter, max_niter = bisection_method_scalar(f, a, b,)

print("Bisection method summary: ")
if ( niter < max_niter ):
    print(f"Convergence reached")
    print(f"Sol, x = {x}")
else:
    print(f"max n.iter reached without convergence")

print(f"residual     : {f(x)}")
print(f"n. iterations: {niter}")
print(f"elapsed time : {time()-t1}")
```

```
Bisection method summary:
Convergence reached
Sol, x = -0.567143440246582
residual     : -2.348157265297246e-07
n. iterations: 20
elapsed time : 0.0006248950958251953
```

Metodo di Newton

Per trovare la soluzione del problema non lineare

$$f(x) = 0\,,$$

il metodo di Newton sfrutta l'espansione in serie troncata al primo grado della funzione $f(x)$, per scrivere

$$0 = f(x^n + \Delta x) \approx f(x^n) + f'(x^n)\Delta x$$

e ottenere l'incremento della soluzione Δx come soluzione del sistema lineare

$$f'(x^n)\Delta x = -f(x^n)$$

e aggiornare la soluzione $x^{n+1} = x^n + \Delta x$.

```
"""
Define newton_method_scalar() function to solve nonlinear scalar equations with Newton
 ↪'s method
"""
```

(continues on next page)

(continued from previous page)

```
def newton_method_scalar(f, df, x=.0, tol=1e-6, max_niter=100):
    """ Function implementing Newton's method for scalar equations """

    res = f(x)
    niter = 0

    # Newton algorithm
    while ( np.abs(res) > tol and niter < max_niter ):
        # Solve linear approximation step, and update solution
        dx = - res / df(x)
        x += dx

        #> Evaluate new residual and n. of iter
        res = f(x)
        niter += 1

    return x, res, niter, max_niter
```

```
""" Use newton_method_scalar() function to solve the example """

# import numpy as np   # already imported

# Parameters of the Newton method, for stopping criteria
# tol = 1e-6          # tolerance on the residual |f(x)| < tol
# max_niter = 10      # max n. of iterations      niter > max_niter
x0 = -1.

t1 = time()
x, res, niter, max_niter = newton_method_scalar(f, df, x=x0)

print("Newton's method summary: ")
if ( niter < max_niter ):
    print(f"Convergence reached")
    print(f"Sol, x = {x}")
else:
    print(f"max n.iter reached without convergence")

print(f"residual     : {f(x)}")
print(f"n. iterations: {niter}")
print(f"elapsed time : {time()-t1}")
```

```
Newton's method summary:
Convergence reached
Sol, x = -0.567143285989123
residual     : 6.927808993140161e-09
n. iterations: 3
elapsed time : 0.0005154609680175781
```

7.2.2 Sistemi di equazioni non lineari

Metodo di Newton

Il metodo di Newton sfrutta l'espansione lineare della funzione $\mathbf{f}(\mathbf{x})$ nell'intorno di un valore $\mathbf{x}$,

$$\mathbf{0} = \mathbf{f}(\mathbf{x} + \mathbf{h}) \simeq \mathbf{f}(\mathbf{x}) + \mathbf{f}'(\mathbf{x})\,\mathbf{h}$$

per costruire un metodo iterativo composto da due passi a ogni iterazione:

- ricerca dell'incremento:

$$\mathbf{f}'(\mathbf{x}^n)\,\mathbf{h}^{(n)} = -\mathbf{f}(\mathbf{x}^{(n)})$$

- aggiornamento della soluzione

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \mathbf{h}^{(n)}\ .$$

Esempio. Il metodo di Newton viene applicato al sistema non lineare

$$\begin{cases} x_0 - x_1 = 0 \\ -x_0^2 + x_1 = -1 \end{cases}$$

che può essere scritto con il formalismo matriciale come

$$\mathbf{0} = \mathbf{f}(\mathbf{x}) = \begin{bmatrix} x_0 - x_1 \\ -x_0^2 + x_1 - 1 \end{bmatrix}\ .$$

La derivata della funzione $\mathbf{f}(\mathbf{x})$, rispetto alla variabile indipendente $\mathbf{x}$,

$$\mathbf{f}'(\mathbf{x}) = \begin{bmatrix} 1 & -1 \\ -2x_0 & 1 \end{bmatrix}$$

può essere rappresentata come una matrice che ha come elemento alla riga i e alla colonna j la derivata della funzione $f_i(\mathbf{x})$ rispetto alla variabile x_j, $[\mathbf{f}']_{ij} = \frac{\partial f_i}{\partial x_j}$, così che l'approssimazione al primo ordine dell'incremento della funzione può essere scritto come

$$\mathbf{f}(\mathbf{x} + \mathbf{h}) - \mathbf{f}(\mathbf{x}) = \mathbf{f}'(\mathbf{x})\,\mathbf{h} + o(||\mathbf{h}||)\ .$$

```
"""
Example. f(x) = np.array([  x[0]    - x[1]
                          -x[0]**2 + x[1] - 1 ])
"""

# Function f and its derivative
f  = lambda x: np.array([ x[0] - x[1], -x[0]**2 + x[1] + 1])
df = lambda x: np.array([[1, -1], [-2*x[0], 1]])
```

```
"""
Define newton_method_scalar() function to solve nonlinear systems of equations with␣
 ↪Newton's method
"""

def newton_method_system(f, df, x=.0, tol=1e-6, max_niter=100):
```

(continues on next page)

(continued from previous page)

```
    """ Function implementing Newton's method for systems of equations """

    res = f(x)
    niter = 0

    # Newton algorithm
    while ( np.linalg.norm(res) > tol and niter < max_niter ):
        # Solve linear approximation step, and update solution
        dx = - np.linalg.solve(df(x), res)
        x += dx

        #> Evaluate new residual and n. of iter
        res = f(x)
        niter += 1

    return x, res, niter, max_niter
```

```
""" Use newton_method_scalar() function to solve the example """

# import numpy as np   # already imported

# Parameters of the Newton method, for stopping criteria
# tol = 1e-6          # tolerance on the residual |f(x)| < tol
# max_niter = 10      # max n. of iterations      niter > max_niter
x0 = np.array([ -1. , 1 ])

t1 = time()
x, res, niter, max_niter = newton_method_system(f, df, x=x0)

print("Newton's method summary: ")
if ( niter < max_niter ):
    print(f"Convergence reached")
    print(f"Sol, x = {x}")
else:
    print(f"max n.iter reached without convergence")

print(f"residual     : {f(x)}")
print(f"n. iterations: {niter}")
print(f"elapsed time : {time()-t1}")
```

```
Newton's method summary:
Convergence reached
Sol, x = [-0.61803399 -0.61803399]
residual     : [ 0.00000000e+00 -2.10942375e-13]
n. iterations: 4
elapsed time : 0.0007855892181396484
```

- **todo** L'algoritmo di Newton trova solo una soluzione del problema. Cercare le altre soluzioni cambiando il tentativo iniziale.
- **todo** … altro?

7.3 Approssimazione di funzioni

7.3.1 Interpolazione

7.3.2 Regressione

7.4 Derivate di funzioni

7.4.1 Differenze finite

Il calcolo della derivata di una funzione $f(x)$ derivabile in un punto x_0 può essere svolto utilizzando l'espansione locale in serie di Taylor di una funzione.

Derivata prima

Usando le espansioni

$$f(x+h) = f(x) + f'(x)\,h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{3!} + o(h^3)$$
$$f(x-h) = f(x) - f'(x)\,h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{3!} + o(h^3)$$

si possono ricavare:

- gli schemi del **primo ordine**

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$
$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h)$$

- lo schema **centrato del secondo ordine**

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

- lo schema non centrato del secondo ordine:

$$f'(x) = \frac{-3\,f(x) + 4\,f(x+h) - f(x+2h)}{2\,h} + O(h^2)\,.$$

Dimostrazione

Usando le espansioni in serie,

$$f(x+h) = f(x) + f'(x)\,h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{3!} + o(h^3)$$
$$f(x+2h) = f(x) + f'(x)\,2h + 2f''(x)\,h^2 + f'''(x)\frac{4}{3}h^3 + o(h^3)$$

si cerca una coppia di coefficienti della combinazione lineare $a_1 f(x+h) + a_2 f(x+2h)$ che annullano il termine di secondo grado.

In particolare è facile dimostrare che una di queste scelte è $\alpha_1 = 4$, $\alpha_2 = -1$, e la combinazione lineare per questi valori diventa,

$$4f(x+h) - f(x+2h) = 3f(x) + 2\,hf'(x) + O(h^3)\,.$$

A questo punto è semplice isolare $f'(x)$ per trovare lo schema numerico desiderato,

$$f'(x) = \frac{-3\,f(x) + 4\,f(x+h) - f(x+2h)}{2\,h} + O(h^2)\,.$$

- …
- schemi di ordine superiore…

Derivata seconda

Usando le stesse espansioni in serie, si può ottenere uno schema del secondo ordine per la derivata seconda

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

Dimostrazione

Usando le espansioni in serie

$$f(x+h) = f(x) + f'(x)\,h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{3!} + f^{iv}(x)\frac{h^4}{4!} + O(h^5)$$
$$f(x-h) = f(x) - f'(x)\,h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{3!} + f^{iv}(x)\frac{h^4}{4!} + O(h^5)$$

si può notare che nella somma che compare al numeratore dello schema numerico si annullano sia il termine di primo grado sia il termine di terzo grado,

$$f(x+h) - 2f(x) + f(x-h) = f''(x)\,h^2 + O(h^4)\,,$$

e quindi lo schema numerico proposto è del secondo ordine,

$$f'(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)\,.$$

```
""" Numerical schemes for derivatives """

df_first_order_left    = lambda f, x, h: (f(x) - f(x-h)) / h
df_first_order_right   = lambda f, x, h: (f(x+h) - f(x)) / h
df_second_order_center = lambda f, x, h: (f(x+h) - f(x-h)) / ( 2. * h )
df_second_order_left   = lambda f, x, h: (  3*f(x) - 4*f(x-h) + f(x-2*h)) / ( 2. * h )
df_second_order_right  = lambda f, x, h: (- 3*f(x) + 4*f(x+h) - f(x+2*h)) / ( 2. * h )

df_fun_dict = {
    '1_left'  : df_first_order_left,
    '1_right' : df_first_order_right,
    '2_center': df_second_order_center,
    '2_left'  : df_second_order_left,
    '2_right' : df_second_order_right
}
```

```
""" Example """
import numpy as np

f = lambda x: 2. * np.exp(- x**2)

x, h = 1., .01

for df_label, df_fun in df_fun_dict.items():
    print(f"Scheme: {df_label.ljust(10)}, value: {df_fun(f,x,h)}" )
```

```
Scheme: 1_left    , value: -1.4788256893130014
Scheme: 1_right   , value: -1.4641117378933477
Scheme: 2_center  , value: -1.4714687136031745
Scheme: 2_left    , value: -1.4716195509633268
Scheme: 2_right   , value: -1.4716121945026028
```

7.5 Integrali

```
""" Import libraries """

import numpy as np
```

7.5.1 Integrazione di Newton-Cotes

Valutazioni di integrali definiti di funzioni $f : [a, b] \in \mathbb{R} \to \mathbb{R}$, continue sull'intervallo. Si veda l'*esempio* per intuire la necessità della continuità della funzione.

- Formula del punto medio
- Formula del trapezio

Metodo di integrazione del punto medio

Il metodo di integrazione del punto medio ricorda la definizione dell'integrale di Riemann (**todo** aggiungere link). Data la funzione $f : [a, b] \in \mathbb{R} \to \mathbb{R}$, e una partizione $P = \{a = x_0 < x_1 < \cdots < x_n = b\}$ dell'intervallo $[a, b]$, l'integrale viene calcolato come la somma dell'area dei rettangoli elementari di lati $\Delta x_k := x_k - x_{k-1}$ e $f(\xi_k)$, con $\xi_k \in [x_{k-1}, x_k]$,

$$\int_{x=a}^{b} f(x)dx \simeq \sum_{k=1:n} f(\xi_k)\,\Delta x_k\,.$$

```
""" Mid-point method """

def integral_rect(f, a, b, n):
    """
    Inputs:
    - f: function
    - a, b: extreme points of the range
    - n: n. of intervals of the range
    """
```

(continues on next page)

(continued from previous page)

```
    # Partition of the range [a,b]
    # uniform partition here; refined algorithms may use non-uniform partitions
    xv = a + ( b - a ) * np.arange(n+1) / n;  xv[-1] = b
    xc = .5 * ( xv[1:] + xv[:-1] )
    dx = xv[1:] - xv[:-1]

    return np.sum( dx * f(xc) )
```

Metodo di integrazione del trapezio

Data la funzione $f : [a, b] \in \mathbb{R} \to \mathbb{R}$, e una partizione $P = \{a = x_0 < x_1 < \cdots < x_n = b\}$ dell'intervallo $[a, b]$, l'integrale viene calcolato come la somma dell'area dei trapezi rettangoli elementari con basi $f(x_{k-1})$, $f(x_k)$ e altezza $\Delta x_k := x_k - x_{k-1}$,

$$\int_{x=a}^{b} f(x)dx \simeq \sum_{k=1:n} \frac{1}{2} \left(f(x_{k-1}) + f(x_k)\right) \Delta x_k \, .$$

```
""" Trapezoid method """

def integral_trapz(f, a, b, n):
    """
    Inputs:
    - f: function
    - a, b: extreme points of the range
    - n: n. of intervals of the range
    """
    # Partition of the range [a,b]
    # uniform partition here; refined algorithms may use non-uniform partitions
    xv = a + ( b - a ) * np.arange(n+1) / n;  xv[-1] = b
    dx = xv[1:] - xv[:-1]

    return np.sum( .5 * dx * ( f(xv[1:]) + f(xv[:-1]) ) )
```

Esempi

Esempio 1.

Si valuta l'integrale

$$\int_{x=0}^{1} x^2 \, dx = \frac{1}{3} \, ,$$

con i metodi del punto medio e del trapezio. La funzione $f(x) = x^2$ è continua ovunque e quindi è continua nell'intervallo $[0, 1]$.

```
f = lambda x: x**2

a, b = 0., 1.
n = 10
```

(continues on next page)

(continued from previous page)

```
# Evaluate integral with the rect and trapz methods
val_rect  = integral_rect(f, a, b, n)
val_trapz = integral_trapz(f, a, b, n)

print(f"Value of the integral, x \in [{a}, {b}] with {n} intervals")
print(f"- mid-point method: {val_rect}")
print(f"- trapezoid method: {val_trapz}")
```

```
Value of the integral, x \in [0.0, 1.0] with 10 intervals
- mid-point method: 0.3325
- trapezoid method: 0.33499999999999996
```

Esempio 2. Necessità della continuità della funzione.

7.5.2 Integrazione di Gauss

L'integrazione di Gauss permette di calcolare in **maniera esatta** l'integrale di una **funzione polinomiale** $p_n(x)$ su un intervallo $[a, b]$, come somma pesata della funzione valuatata in alcuni punti dell'intervallo,

$$\int_a^b p^{(n)}(x)dx = \sum_g w_g f(x_g) \, ,$$

per un numero di nodi di Gauss, n_{Gauss} che soddisfa la relazione

$$n < 2n_{Gauss} - 1 \, .$$

Per motivi di generalizzazione dell'algoritmo, nella definizione dei **pesi** w_i e dei **nodi di Gauss** x_i, l'integrale viene riportato all'integrale su un intervallo di riferimento, tramite una trasformazione di coordinate. Per domini 1D, l'intervallo di riferimento per la quadratura di Gauss è l'intervallo $\xi = [-1, 1]$ e il cambio di variabili è

$$x = \frac{a+b}{2} + \frac{b-a}{2}\,\xi \, ,$$

così che l'integrale originale può essere scritto come

$$\int_{x=a}^{b} p^{(n)}(x)dx = \int_{\xi=-1}^{1} p^(n)(x(\xi))\frac{dx}{d\xi}d\xi = \\ = \frac{b-a}{2}\int_{\xi=-1}^{1} p^{(n)}(x(\xi))d\xi = \frac{b-a}{2}\sum_g w_g\, p^{(n)}\left(x(\xi_g)\right)$$

```
""" Gauss integration """

# Dict of Gauss weights and nodes on the reference interval [-1,1] for 1D integration
gauss_nw = { 1: {'nodes'   : [ 0. ],
                 'weights' : [ 2. ]},
             2: {'nodes'   : [ -1./np.sqrt(3.), 1./np.sqrt(3.) ],
                 'weights' : [ 1., 1.]},
             3: {'nodes'   : [ 0., -np.sqrt(3./5.), np.sqrt(3./5.) ],
                 'weights' : [ 8./9., 5./9., 5./9. ]},
}
```

(continues on next page)

(continued from previous page)

```
def gauss_int_1(f, n, a=-1, b=1, gauss_nw=gauss_nw):
    """
    Integral of the function f(x)
    over a physical domain, x \in [a, b],
    using n number of Gauss nodes
    """
    # Cap n.nodes to the max n.nodes stored in gauss_nw dict
    n = np.min([n, np.max(list(gauss_nw.keys()))])

    # Gauss nodes (from reference to actual domain) and weights
    xg = .5 * ( a + b + ( b - a ) * np.array(gauss_nw[n]['nodes']) )
    wg = np.array(gauss_nw[n]['weights'])

    return .5 * (b-a) * np.sum( wg * f(xg) )


def gauss_int_n(f, n_gauss, a, b, n_elems):
    """
    Integral of the function f(x)
    over a physical domain, x \in [a, b], (uniformly) splitted in n_elems
    using n number of Gauss nodes per each elem
    """
    # Partition of the interval [a,b]
    xv = a + ( b - a ) * np.arange(n_elems+1) / n_elems;  xv[-1] = b

    return np.sum([ gauss_int_1(f, n_gauss, xv[i], xv[i+1]) for i in np.arange(n_
↪elems) ])
```

Esempio 1 - integrazione di Gauss.

Si valuta l'integrale

$$\int_{x=0}^{1} x^2 \, dx = \frac{1}{3} \, ,$$

con il metodo di integrazione di Gauss. Si chiede di:

- osservare che l'integrazione è esatta, a meno degli arrotondamenti dovuti all'aritmetica finita, poiché la funzione integranda è di grado 2, e il numero di nodi di Gauss è $n_{Gauss} = 3$; questo è vero indipendentemente dal numero di sotto-intervalli;
- cambiare le funzioni e il numero di nodi usati nelle funzioni per l'integrazione di Gauss per verificare che l'integrazione è esatta per funzioni polinomiali di grado $2n_{Gauss} - 1$.

```
""" Evaluate integrals using Gauss integration """

f = lambda x: x**2
a, b = 0., 1.

# Gauss nodes, and n.of elements
n_gauss = 3
n_elems = 10

val_1 = gauss_int_1(f, n_gauss, 0., 1.)
```

(continues on next page)

(continued from previous page)

```
val_n = gauss_int_n(f, n_gauss, 0., 1., n_elems)

print(f"Value of the integral, x \in [{a}, {b}] using Gauss integration methods with
 ↪{n_gauss} nodes per elem")
print(f"- using one elem: {val_1}")
print(f"- using {n_elems} elems: {val_n}")
```

```
Value of the integral, x \in [0.0, 1.0] using Gauss integration methods with 3␣
 ↪nodes per elem
- using one elem: 0.33333333333333337
- using 10 elems: 0.33333333333333337
```

7.6 Equazioni differenziali ordinarie

7.6.1 Problemi di Cauchy ai valori iniziali

Approccio a un problema di Cauchy di ordine n

Un problema di Cauchy di ordine n

$$\begin{cases} F(y^{(n)}(x), y^{(n-1)}(x), \dots, y'(x), y(x), x) = 0 \\ y(x_0) = y^0 \\ y'(x_0) = y'^0 \\ \dots \\ y^{(n-1)}(x_0) = y^{(n-1),0} \end{cases}$$

con funzione incognita $y(x) : D \in \mathbb{R} \to \mathbb{R}$, può essere riscritto come un problema di *"ordine 1"* per la funzione incognita $\mathbf{z}(x) : D \in \mathbb{R} \to \mathbb{R}^n$, definita come

$$\mathbf{z}(x) = (z_0(x), z_1(x), \dots z_{n-1}(x)) := (y(x), y'(x), \dots, y^{(n-1)}(x)) \, .$$

Esplicitando le relazioni tra le componenti di $\mathbf{z}(x)$ e le derivate della funzione $y(x)$, $z_k(x) = y^{(k)}(x) = y^{(k-1)'}(x) = z'_{k-1}(x)$, il problema di Cauchy può essere riformulato come

$$\begin{cases} z'_0 - z_1 = 0 \\ z'_1 - z_2 = 0 \\ \dots \\ z'_{n-2} - z_{n-1} = 0 \\ F(z'_{n-1}(x), z_{n-1}(x), \dots, z_1(x), z_0(x)) = 0 \, , \end{cases} \quad , \quad \text{i.c.} \quad \begin{cases} z_0(x_0) = y^0 \\ z_1(x_0) = y'^0 \\ \dots \\ z_{n-1}(x_0) = y^{(n-1),0} \end{cases}$$

che può essere riscritto con il formalismo vettoriale come

$$\begin{cases} \mathbf{F}(\mathbf{z}'(x), \mathbf{z}) = \mathbf{0} \\ \mathbf{z}(x_0) = \mathbf{z}_0 \end{cases}$$

Caratteristiche (cenni)

- accuratezza, consistenza, convergenza
- stabilità: 0-, A- condizionata e incondizionata

Schemi numerici

Schemi numerici a un passo

Schemi numerici multi-step

7.6.2 Problemi al contorno

Differenze finite

Elementi finiti

Volumi finiti

7.7 Ottimizzazione

Le tecniche di ottimizzazione sono alla base di molti metodi di interesse, dall'approssimazione di funzioni, alla regolazione e controllo, agli algoritmi usati in intelligenza artificiale

Part III

Introduzione ai metodi in statistica e AI

CHAPTER

EIGHT

INTRODUZIONE AI METODI IN AI

In questa sezione si descriveranno alcuni metodi utilizzati nelle applicazioni di **machine learning**.

Approccio. Alla completezza e al rigore dello sviluppo teorico dei metodi, viene preferita una descrizione dei metodi tramite esempi e applicazioni.

Benché questo possa essere un approccio ad alto rischio di creazione di utenti acritici di strumenti che non comprendono, la speranza è di mitigare questo rischio con continui moniti a prestare attenzione e un supporto del lettore da parte di gente con un minimo di esperienza.

Argomenti. In questa sezione verranno presentati alcuni metodi e applicazioni del **machine learning**, che possono essere classificate in tre grandi classi di apprendimento:

- SL, supervised learning: regression and classification
- UL, unsupervised learning: clustering
- ML, machine learning: control

Dopo aver presentato le tecniche classiche, verranno introdotte le reti neurali, gli algoritmi fondamentali che hanno permesso un uso pratico ed efficiente di reti profonde, e alcune architetture fondamentali di reti neurali.

Le applicazioni verranno affrontate inizialmente con approcci classici, affidandoci alla libreria sci-kit, e successivamente con tecniche “deep” grazie alla libreria PyTorch.

Part IV

Supporto tecnico

CHAPTER

NINE

SUPPORTO TECNICO

- git
- Servizi Google
- TeX