Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Progetto 1: Sistemi lineari con matrici sparse, simmetriche e definite positive

Corso: Metodi del Calcolo Scientifico

Relazione di:
Alessandro Capelli 816302
Damiano Dovico 816682
Davide Bucci 816067
Emilio Brambilla 816538

Anno Accademico 2019-2020

Indice

1 Introduzione

1.1 Scopo del progetto

Questo progetto ha lo scopo di studiare l'implementazione del **metodo di Choleski**, in ambiente di programmazione open source, per la risoluzione di sistemi lineari per **matrici sparse, simmetriche e definite positive** e di confrontarla con l'implementazione in ambiente MATLAB. L'obiettivo consiste nell'analisi di alcune metriche che consentano di valutare una scelta ponderata dell'ambiente più idoneo in base al contesto aziendale o individuale nel quale ci si trova: sia come ambiente di sviluppo, sia come sistema operativo. In particolare, dopo aver implementato il metodo di Choleski per la risoluzione di sistemi lineari, è stato analizzato il comportamento di tale implementazione in ambiente MATLAB (come ambiente proprietario) e ambiente Python (come ambiente open source), sia su sistema Windows che su sistema Linux; i confronti sono avvenuti in termini di **tempo**, **accuratezza** (misurata come errore relativo), **memoria utilizzata**, **facilità d'uso** e **documentazione**.

1.2 Matrici sparse

Proposizione 1. *Le **matrici sparse** sono matrici con un grande numero di elementi uguali a zero.*

Le matrici sparse hanno la peculiarità di essere memorizzate in modo compatto, tenendo solo il conto degli elementi diversi da zero (e, g,. memorizzando la posizione (i, j) e il valore a_{ij}) e per questo sono spesso utilizzate nei problemi del calcolo scientifico per ottimizzare la ricerca di soluzioni.

1.3 Decomposizione di Cholesky

La decomposizione di Cholesky è un metodo di fattorizzazione matriciale semplice ed efficiente, applicabile a matrici generiche $A \in \mathbb{R}^{n\times n}$ che sono:

1. **simmetriche:** $A = A^t$
2. **definite positive:** $y^t Ay > 0, \forall\, y \in \mathbb{R}^n \setminus \{0\}$

Teorema 1. *Sia $A \in \mathbb{R}^{n\times n}$ una matrice simmetrica e definita positiva. Allora esiste una matrice $R \in \mathbb{R}^{n\times n}$ triangolare superiore tale che $A = R^t R$. La R è unica se si aggiunge l'ulteriore richiesta che gli elementi sulla diagonale principale siano positivi.*

Si noti che la complessità computazionale della decomposizione di Cholesky è $O(n^3)$.

1.4 Sistemi lineari

Dato un sistema lineare $Ax = b$, in cui la matrice A gode delle proprietà necessarie per la decomposizione, è applicabile il metodo di Cholesky per risolverlo. Una volta ottenuta la matrice R risulta immediata la risoluzione del sistema lineare come segue: si sostituisce alla matrice A la sua fattorizzazione e si considera il sistema $R^t Rx = b$; successivamente si risolvono in ordine un sistema triangolare inferiore e poi uno triangolare superiore:

- $R^t y = b$

- $Rx = y$

I sistemi vengono risolti rispettivamente tramite sostituzione in avanti e all'indietro. In particolare, i sistemi lineari utilizzati nei solutori diretti per matrici simmetriche e definite positive sparse assumono la seguente forma: $Ax = b$ con b scelto in modo tale che la soluzione esatta sia il vettore $x_e = [1 \dots 1]$ (quindi $b = Ax_e$).

1.5 Matrici utilizzate

Definizione formale del problema:

- **Input:** matrice M sparsa, simmetrica e definita positiva
- **Dataset:** matrici presenti in "SuiteSparce Matrix Collection" [1]
  - Flan_1565 [2]:
    - numero righe e colonne: 1564794
    - numero di elementi diversi da zero: 114165372
    - percentuale di riempimento: $\approx$ 1.37 %
  - StocF-1465 [3]:
    - numero righe e colonne: 1465137
    - numero di elementi diversi da zero: 21005389
    - percentuale di riempimento: $\approx$ 6.98 %
  - cfd2 [4]:
    - numero righe e colonne: 123440
    - numero di elementi diversi da zero: 3085406
    - percentuale di riempimento: $\approx$ 4.00 %
  - cfd1 [5]:
    - numero righe e colonne: 70656
    - numero di elementi diversi da zero: 1825580
    - percentuale di riempimento: $\approx$ 3.87 %
  - G3_circuit [6]:
    - numero righe e colonne: 1585478
    - numero di elementi diversi da zero: 7660826
    - percentuale di riempimento: $\approx$ 20.70 %
  - parabolic_fem [7]:
    - numero righe e colonne: 525825
    - numero di elementi diversi da zero: 3674625
    - percentuale di riempimento: $\approx$ 14.31 %
  - apache2 [8]:
    - numero righe e colonne: 715176
    - numero di elementi diversi da zero: 4817870
    - percentuale di riempimento: $\approx$ 14.84 %

- shallow_water1 [9]:
  - numero righe e colonne: 81920
  - numero di elementi diversi da zero: 327680
  - percentuale di riempimento: ≈ 25 %
- ex15 [10]:
  - numero righe e colonne: 6867
  - numero di elementi diversi da zero: 98671
  - percentuale di riempimento: ≈ 6.96 %

1.6 Ambienti selezionati

La decisione fondamentale, su cui si sono basate le successive analisi, è stata quella inerente alla scelta dell'ambiente open source da comparare con la controparte proprietaria MATLAB. La scelta è stata dettata da un'analisi dei costi-benefici derivanti dall'utilizzo di ambienti specializzati nel calcolo numerico e nella gestione di grosse quantità di dati (rappresentate dalle matrici prese in esame). In particolare, l'attenzione è stata riposta su Python, che mette a disposizione un linguaggio ad "alto livello", numerose librerie ottimizzate per il calcolo numerico e per cui si trovano enormi quantità di informazioni e documentazione. La peculiarità di Python consiste nella velocità di apprendimento, molto più intuitivo ed immediato rispetto a linguaggi con sintassi più articolate (e.g. C, C++). Altre caratteristiche degne di nota per lo studio effettuato sono rappresentate dalla facilità di installazione, gestione dei pacchetti ed interoperabilità dei codici sorgenti. Come svantaggio in termini di velocità di esecuzione è necessario riportare la natura interpretata del linguaggio (quindi non compilata), che ha portato ad avere un confronto più equo e trasparente con MATLAB, anch'esso interpretato (figura 1 per un confronto tra gli ecosistemi); a questa nota è necessario specificare l'esistenza di compilatori JIT [11], in grado di rendere più efficiente l'esecuzione di programmi Python (in particolar modo quelli inerenti all'elaborazione di calcoli matematici). Un altro ambiente considerato tra i possibili candidati è stato R [12], ma dopo un primo studio è emersa una problematica relativa alla carenza di librerie riguardanti la decomposizione di Cholesky rendendo poco adatto l'ambiente in questione. Pertanto, Python è risultata la scelta ottimale.

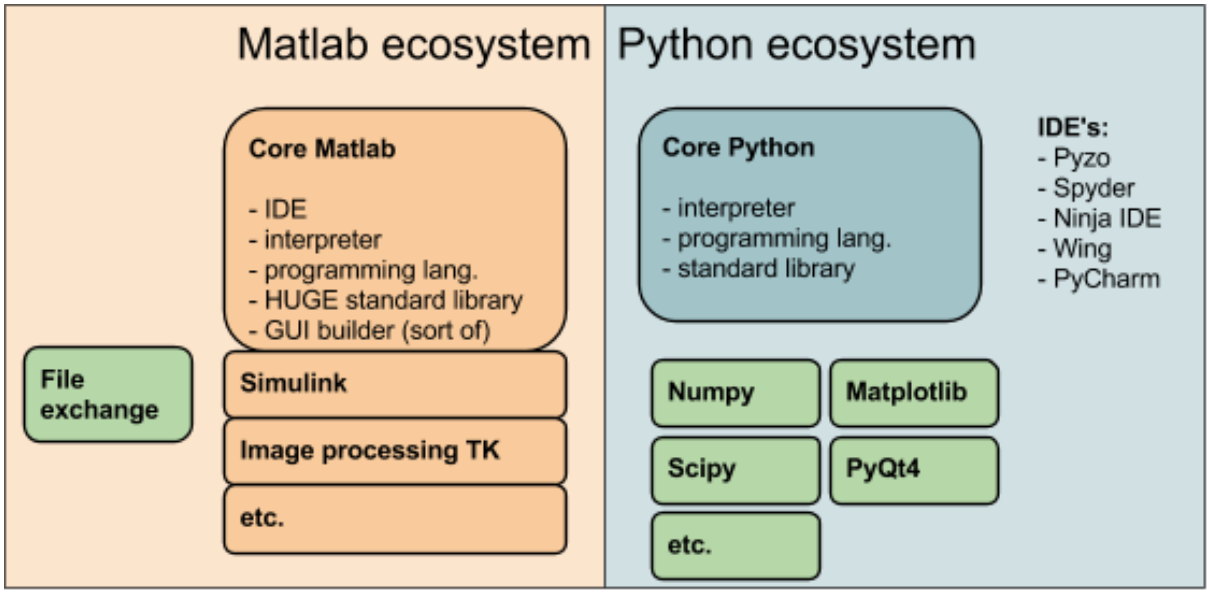

Figura 1: Confronto tra Python e MATLAB

1.7 Metriche

Per ogni matrice considerata, sono state calcolate le seguenti metriche:

- **tempo:** è stato valutato il tempo necessario a calcolare la soluzione x associata al sistema lineare. È misurato in millisecondi e successivamente convertito in secondi per rendere tale dato più leggibile e facilmente interpretabile
- **errore relativo:** è stato calcolato l'errore relativo tra la soluzione ottenuta x e la soluzione esatta x_e. Formula: $\frac{||x-x_e||_2}{||x_e||_2}$ con $||v||_2$ che rappresenta la norma euclidea del vettore v
- **memoria:** è stata calcolata la memoria (RAM + swap) necessaria per risolvere il sistema, determinata come l'aumento della dimensione del programma in memoria (dalla lettura della matrice alla risoluzione del sistema); misurata in byte e successivamente convertita MiB (mebibyte = 2^{20} byte)

Tali metriche sono alla base dei confronti, delle analisi e delle conclusioni finali e rappresentano i dati principali su cui si sono svolti gli studi degli ambienti programmativi e dei sistemi operativi che le hanno generate.

2 Specifiche hardware

2.1 Introduzione

Una fase fondamentale del progetto è stata la scelta dell'hardware da utilizzare per i calcoli. La richiesta era quella di trovare una macchina, preferibilmente una non virtualizzata (per evitare l'acquisizione di dati falsati a livello di integrazione software con l'ambiente host), che potesse eseguire sia un sistema operativo Linux che un sistema Windows (entrambi supportano Python versione 3.7 [13] e MATLAB versione R2020a [14]). Dopo aver valutato le macchine a disposizione dei membri del gruppo, è stato preso in considerazione come macchina di riferimento un portatile con processore Intel i3, che presentava tutti i requisiti necessari a poter eseguire in modo diretto i test e per poter acquisire i dati nella maniera più trasparente possibile.

Inoltre, sono state considerate ulteriori architetture per consentire una più ampia valutazione del caso in analisi: un ambiente virtualizzato ed un ulteriore sistema operativo.

- Azure [15] come ambiente virtualizzato (sia per Windows che per Linux, mantenendo le medesime caratteristiche hardware sottostanti)
- MAC OS (Darwin)

Chiaramente i dati derivanti dalle diverse macchine considerate non sono direttamente confrontabili tra loro in quanto hanno caratteristiche computazionali molto differenti, i dati principali sono quelli prodotti dalla macchina di riferimento 2.2.

2.2 Macchina di riferimento

Le specifiche tecniche relative alla **macchina di riferimento** sono le seguenti:

- **Sistema:** Asus K53SJ
- **SO:** Ubuntu 19.10 - Windows 10 Education
- **CPU:** Intel Core i3 2310M - 2.10 GHz Dual-Core
- **RAM:** 8 GB 1333 MHz DDR3
- **Disco rigido:** 256 GB SSD

2.3 Virtual machine

Le specifiche tecniche relative alla **virtual machine** sono le seguenti:

- **Sistema:** server Azure virtualizzato
- **SO:** CentOS 8.1.1911 - Windows 10 Pro
- **CPU:** Intel XEON E5-2673 v4 (Broadwell) - 2.3 GHz Dual-Core
- **RAM:** 28 GB
- **Disco rigido:** 128 GB SSD Premium di Azure

2.4 MacOS

Le specifiche tecniche relative al computer **Darwin** sono le seguenti:

- **Sistema:** MacBook Pro (15-inch, 2017)
- **SO:** macOS Catalina (10.15.4)
- **CPU:** Intel Core i7-7820HQ - 2.9 GHz Quad-Core
- **RAM:** 16 GB 2133 MHz LPDDR3
- **Disco rigido:** 500 GB SSD

3 Implementazione Python

3.1 Introduzione

L'implementazione Python ha portato alla stesura di un codice che genera un file in formato CSV [16] contenente tutte le informazioni richieste e necessarie ad un successivo assemblaggio per l'esportazione nei relativi grafici.

Le funzioni principali consistono nel caricamento delle matrici e nella loro successiva elaborazione in termini di sistema lineare associato, in modo sequenziale, ponendo attenzione ad evitare il fenomeno del "fill-in". A tal proposito è stata utilizzata la funzione **loadmat** presente nella libreria scipy.io per il caricamento in memoria della matrice, in concomitanza con **sparse2cvxopt** presente nel modulo matrix_utilities di cvxpy.interface per sfruttare la sparsità della matrice data in input. Per il calcolo delle risorse utilizzate in termini di memoria (RAM + swap) sono state utilizzate delle librerie cross-platform (psutil [17] e memory_profiler [18]) per garantire l'interoperabilità del codice e l'integrità dei risultati ottenuti su piattaforme differenti.

3.2 Librerie

Inizialmente si è optato per la libreria **Scipy** per la risoluzione di sistemi lineari, affiancata dalla libreria **Numpy** per quanto riguarda la gestione matematica degli array multidimensionali. Nello specifico, è disponibile una libreria denominata **scipy.linalg** [19] che implementa funzioni di algebra lineare in modo ottimizzato; al suo interno si possono trovare due metodi, che consentono l'utilizzo della decomposizione di Cholesky per risolvere sistemi lineari:

- **scipy.linalg.cho_factor**
- **scipy.linalg.cho_solve**

La prima funzione computa, data una matrice simmetrica e definita positiva A, la relativa decomposizione di Cholesky e restituisce una matrice triangolare superiore/inferiore R, in base ai parametri che vengono forniti in input. Successivamente il metodo cho_solve, presi in input la matrice triangolare R (precedentemente ottenuta) e l'array dei termini noti b, restituisce la soluzione del sistema lineare $Ax = b$, ovvero l'array x. Tuttavia, a seguito di questa prima implementazione testata su matrici generate in modo randomico, è stato osservato che i metodi forniti dalla libreria Scipy permettono di operare solo su matrici di tipo denso. Effettuando tale scelta si otterrebbe una perdita di efficienza data dalla sparsità delle matrici prese in considerazione (rendendo computazionalmente ingestibili i calcoli). In questo specifico caso, avendo la necessità di operare su matrici di tipo sparso, è stato quindi necessario trovare una valida alternativa a questa prima implementazione.

Una seconda libreria presa in considerazione è stata **scikit-sparse** [20], la quale implementa efficientemente le operazioni di algebra lineare per matrici sparse, simmetriche e definite positive; più precisamente la libreria in questione altro non è che un'estensione della precedente (Scipy). Al suo interno sono disponibili due metodi principali per poter risolvere un sistema lineare tramite la fattorizzazione di Cholesky:

- **cholesky(A):** calcola la decomposizione di Cholesky per la matrice A data in input
- **factor(b):** restituisce la soluzione del sistema lineare $Ax = b$

La libreria in questione presenta tuttavia un grosso vincolo, ovvero la difficoltà di installazione in ambiente Windows; tale limitazione, sommata alla scarsa documentazione associata, hanno reso tale libreria escludibile dal progetto. Per completezza nozionistica si cita una guida trovata online [21] in cui sono presenti vari punti da seguire per poter effettuare un'installazione corretta del pacchetto, ma al termine delle operazioni persistevano comunque svariati problemi: in alcuni casi la libreria non veniva importata correttamente, altre volte emergevano problematiche legate all'ambiente di utilizzo.

A seguito delle precedenti problematiche, è stata effettuata un'ulteriore ricerca, con l'aggiunta del vincolo di semplicità di installazione nei vari ambienti, di completezza di documentazione associata e di semplicità di utilizzo della libreria stessa, mantenendo comunque le medesime funzionalità. La ricerche hanno portato ad un'ulteriore libreria, utilizzata effettivamente nell'implementazione finale dell'algoritmo: **cvxopt**. Essa fornisce dei metodi per effettuare calcoli su matrici dense e sparse ed è periodicamente aggiornata e documentata [22]. Nello specifico è stato utilizzato il pacchetto **cvxopt.cholmod**, cioè un'interfaccia della libreria **CHOLMOD** [23]. CHOLMOD è una libreria compatibile sia con Windows che con Linux altamente performante per computare la fattorizzazione di Cholesky in maniera ottimale su matrici sparse, simmetriche e definite positive [24], nonché parte integrante della famosa libreria di algebra lineare **SuiteSparse** (implementata in diversi ambienti di sviluppo, proprietari ed open source). Nello specifico la funzione protagonista nella risoluzione dei sistemi lineari tramite Cholesky è **cvxopt.cholmod.splinsolve()**. Questa funzione ha come parametri di input una matrice sparsa A ed il relativo vettore dei termini noti b e, a seguito della computazione, restituisce il vettore delle soluzioni x.

3.3 Codice

```
import numpy as np
import csv
import time
import platform
from scipy.io import loadmat
from cvxpy.interface import matrix_utilities
from cvxopt import sparse, matrix, cholmod
from scipy import linalg
import psutil
from memory_profiler import memory_usage

# Global variables
INPUT_DIRECTORY = '/Users/'
MATRICES_NAMES = ['ex15', 'shallow_water1', 'apache2', 'parabolic_fem',
    'G3_circuit', 'cfd1', 'cfd2', 'StocF-1465', 'Flan_1565']
OUTPUT_PATH = INPUT_DIRECTORY + '/data_python_' + platform.system().
    lower() + '.csv'
CSV_HEADER = ['Environment', 'System', 'Matrix name', 'Elapsed time (s)
    ', 'Memory (MB)', 'Relative error']

def resolve(matrix_name):
    A = matrix_utilities.sparse2cvxopt(loadmat(INPUT_DIRECTORY +
    matrix_name)['Problem']['A'][0][0])

    start_memory = psutil.swap_memory()[1] / (1024 * 1024)

```

```
23      xe = matrix(np.ones([A.size[0], 1]))
24      b = sparse(A * xe)
25
26      end_memory = max(memory_usage((cholmod.splinsolve, (A, b)))) + (
        psutil.swap_memory()[1] / (1024 * 1024))
27
28      start_time = time.time()
29
30      x = cholmod.splinsolve(A, b)
31
32      end_time = time.time()
33
34      error = linalg.norm(x - xe) / linalg.norm(xe)
35
36      return (end_time - start_time), (end_memory - start_memory), error
37
38  def cholesky_python():
39      with open(OUTPUT_PATH, 'w', newline='\n') as data:
40          write_data = csv.writer(data, quoting=csv.QUOTE_ALL)
41          write_data.writerow(CSV_HEADER)
42
43          for matrix_name in MATRICES_NAMES:
44              time, memory, error = resolve(matrix_name)
45
46              write_data.writerow(['Python', platform.system(),
        matrix_name, time, memory, error])
47              print("Environment: {}\nSystem: {}\nMatrix: {}\nElapsed
        time: {} s\nMemory: {} MB\nRelative error: {}\n".format('Python',
        platform.system(), matrix_name, time, memory, error))
48
49  if __name__ == '__main__':
50      cholesky_python()
```

Listing 1: cholesky_python.py

Per calcolare in maniera esatta l'utilizzo di memoria (RAM + swap) utilizzata dalle matrici più grosse (Flan_1565 e StocF-1465) è stato necessario implementare uno script ad hoc che, attraverso la programmazione multithreading, permettesse di monitorare in tempo reale le risorse allocate, tenendo conto in particolar modo della memoria swap:

```
from multiprocessing.pool import Pool
import numpy as np
from cvxopt import cholmod, matrix, sparse
from cvxpy.interface import matrix_utilities
import time
from scipy import io
import psutil

def memory_watch():
    start = str(int(time.time()))
    virtual_start = (psutil.virtual_memory().used) / (1024 * 1024)
    swap_start = (psutil.swap_memory().used) / (1024 * 1024)
    sampling_frequence = 10

    while True:
        virtual = (psutil.virtual_memory().used) / (1024 * 1024) -
    virtual_start
        swap = (psutil.swap_memory().used) / (1024 * 1024) - swap_start

        memory = virtual + swap

        with open(start + '.txt', 'a') as txt:
            txt.write(str(memory) + ', ')

        time.sleep(sampling_frequence)

def main():
    matrices = ['StocF-1465.mat', 'Flan_1565.mat']

    for matrix_name in matrices:
        A = matrix_utilities.sparse2cvxopt(io.loadmat(matrix_name)['
    Problem']['A'][0][0])
        xe = matrix(np.ones([A.size[0], 1]))
        b = sparse(A * xe)

        pool = Pool(processes=1)
        pool.apply_async(memory_watch)
        time.sleep(5)

        cholmod.splinsolve(A, b)

        pool.terminate()
        pool.join()

if __name__ == '__main__':
    main()
```

Listing 2: cholesky_python_memory.py

4 Implementazione MATLAB

4.1 Introduzione

L'implementazione MATLAB ha portato alla stesura di un codice che genera un file in formato CSV [16] contenente tutte le informazioni richieste e necessarie ad un successivo assemblaggio per l'esportazione nei relativi grafici.

Le funzioni principali consistono nel caricamento delle matrici e nella loro successiva elaborazione in termini di sistema lineare associato, in modo sequenziale, porgendo (come in Python) particolare attenzione al fenomeno del "fill-in"; a tal proposito è stata utilizzata la funzione **load** presente nella libreria standard di MATLAB per il caricamento in memoria della matrice. Per il calcolo delle risorse utilizzate in termine di memoria (RAM + swap) è stata utilizzata la funzione **whos** (presente nella libreria standard di MATLAB) mentre per il calcolo del tempo impiegato è stata utilizzata la funzione **tic** (con la controparte **toc**), anch'essa presente nella libreria standard.

4.2 Librerie

Per implementare la risoluzione di sistemi lineari con il metodo di Cholesky in ambiente MATLAB è stato utilizzato il metodo **mldivide** (associato all'operatore "\"). Tale funzione consente di risolvere sistemi scritti nella forma $Ax = b$ semplicemente utilizzando la seguente sintassi: $x = A\backslash B$.

La scelta è stata dettata dalla facilità di utilizzo, dall'interoperabilità tra sistemi operativi, dalla documentazione esaustiva e dalle alte prestazioni misurate. Purtroppo l'utilizzo di mldivide non garantisce l'esecuzione del metodo di Cholesky: il workflow intrapreso da mldivide [25] cerca di ottimizzare l'elaborazione scegliendo il metodo risolutivo più opportuno prima di applicarlo.

Un altro problema relativo all'utilizzo di mldivide si riscontra nella profilazione della memoria: utilizzando tale metodo è complicato misurare in modo esatto la quantità di memoria impiegata per il calcolo, poiché l'allocazione e de-allocazione degli elementi avvengono all'interno della funzione. Per ovviare a queste problematiche è stato necessario applicare il metodo **chol** [26] in concomitanza a mldivide. Il procedimento (per analisi più approfondite: benchmark [27]) sfrutta la matrice di permutazione, ovvero un preordinamento della matrice sparsa data in input, restituita dal metodo chol rendendo efficienti i calcoli:

- $R, P = chol(A)$
- $x = P \times (R' \ \backslash \ (R \ \backslash \ (P' \times b)))$

Tuttavia utilizzando questo metodo sorgono delle problematiche legate ai tempi di calcolo e all'occupazione di memoria, le quali sono nettamente superiori rispetto all'applicazione pura di mldivide. Citando una discussione di interesse [28] sul forum ufficiale di MATLAB in cui vengono proposti i confronti tra le metodologie considerate, il problema è da attribuirsi al fatto che l'applicazione di $R = chol(A)$ per matrici sparse tende ad avere un grande numero di **fill-in** che conseguentemente porta ad avere le sopracitate problematiche. Utilizzando mldivide, il problema viene evitato permutando inizialmente gli elementi di A consentendo di arginare efficacemente questa problematica. Riassumendo: il metodo misto utilizzato nel progetto evita il fill-in, necessita di più tempo di

elaborazione rispetto a mldivide puro ma è più efficiente rispetto a chol utilizzato senza matrice di permutazione.

4.3 Codice

```
function [] = cholesky_matlab()
    if(ispc)
        system = 'Windows';
    elseif(ismac)
        system = 'Darwin';
    else
        system = 'Linux';
    end

    % Global variables
    INPUT_DIRECTORY = '/Users/';
    MATRICES_NAMES = ["ex15" "shallow_water1" "apache2" "parabolic_fem" "G3_circuit" "cfd1" "cfd2" "StocF-1465" "Flan_1565"];
    OUTPUT_PATH = strcat(INPUT_DIRECTORY, 'data_matlab_', lower(system), '.csv');
    CSV_HEADER = {'Environment' 'System' 'Matrix name' 'Elapsed time (s)' 'Memory (MB)' 'Relative error'};
    CSV_HEADER = [CSV_HEADER;repmat({','}, 1, numel(CSV_HEADER))];
    CSV_HEADER = cell2mat(CSV_HEADER(:)');
    CSV_HEADER = CSV_HEADER(1 : end-1);

    csv_file = fopen(OUTPUT_PATH, 'w');
    fprintf(csv_file, '%s\n', CSV_HEADER);

    for matrix = MATRICES_NAMES
        [time, memory, error] = resolve(strcat(INPUT_DIRECTORY, matrix));

        info = ['MATLAB' system matrix time memory error];
        [rows, ~] = size(info);
        for i=1 : rows
              fprintf(csv_file, '%s,', info{i,1:end-1});
              fprintf(csv_file,'%s\n', info{i,end});
        end

        sprintf('Environment: MATLAB\nSystem: %s\nMatrix: %s\nElapsed time: %.8f s\nMemory: %.2f MB\nRelative error: %d\n', system, matrix, time, memory, error)
    end

    fclose(csv_file);
```

```
36 end
37
38 function [time, memory, error] = resolve(matrix_path)
39     M = load(matrix_path);
40     A = M.Problem.A;
41     clear M;
42
43     start_memory = whos;
44     start_memory = sum([start_memory.bytes]);
45
46     xe = ones(length(A), 1);
47     b = A * xe;
48
49     tic
50
51     [R, ~, P] = chol(A, 'lower');
52     x = P * (R' \ (R \ (P' * b)));
53     % x = A \ b; % mldivide
54
55     time = toc;
56
57     error = norm(x - xe) / norm(xe);
58
59     end_memory = whos;
60     end_memory = sum([end_memory.bytes]);
61     memory = (end_memory - start_memory) / (1024 * 1024);
62 end
```

Listing 3: cholesky_matlab.m

5 Test ed analisi dei risultati

In questa sezione vengono presentati i test eseguiti con associati i relativi risultati ottenuti, in particolar modo prestando attenzione alle seguenti metriche in termini dei diversi ambienti considerati:

Tempi	Memoria	Software	Ambiente
Secondi	MebiByte	MATLAB/Python	OS

È importante sottolineare due aspetti molto importanti:

- In primo luogo tutti i risultati riportati nelle tabelle fanno riferimento ad una **media** dei dati stimati
- I grafici riportano risultati in scala **semi-logaritmica**, questo comporta un apparente vicinanza dei dati anche laddove essi risultano distanti

5.1 Macchina di riferimento

Questa serie di test sono stati effettuati sulla macchina di riferimento, cioè un portatile avente dual boot le cui specifiche sono approfondite nella sezione 2.

5.1.1 Windows

Una prima serie di test di confronto tra MATLAB e Python sono stati eseguiti con sistema operativo Windows, i risultati ottenuti sono osservabili nella tabella 1.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Python	554	9232	6.79E-08
MATLAB	1142	13815	7.39E-08

Tabella 1: Tempo, memoria ed errore medi su Windows

Dai dati si può subito notare una netta superiorità di Python rispetto alla controparte MATLAB: mediamente il tempo computazionale di quest'ultimo è doppio, l'occupazione di memoria è superiore di circa 4000 MebiByte e l'errore relativo è lievemente superiore. I risultati sono anche riportati sotto forma di grafico, osservabili in figura 2, che consentono di osservare precisamente l'andamento dei software. Si noti in particolare che il comportamento dei due ambienti è simile per quanto riguarda la computazione effettuata su tutte le matrici considerate; ciò è in linea con l'uso delle risorse monitorato sia in modo automatizzato tramite script, sia in modo manuale tramite l'osservazione dei processi coinvolti.

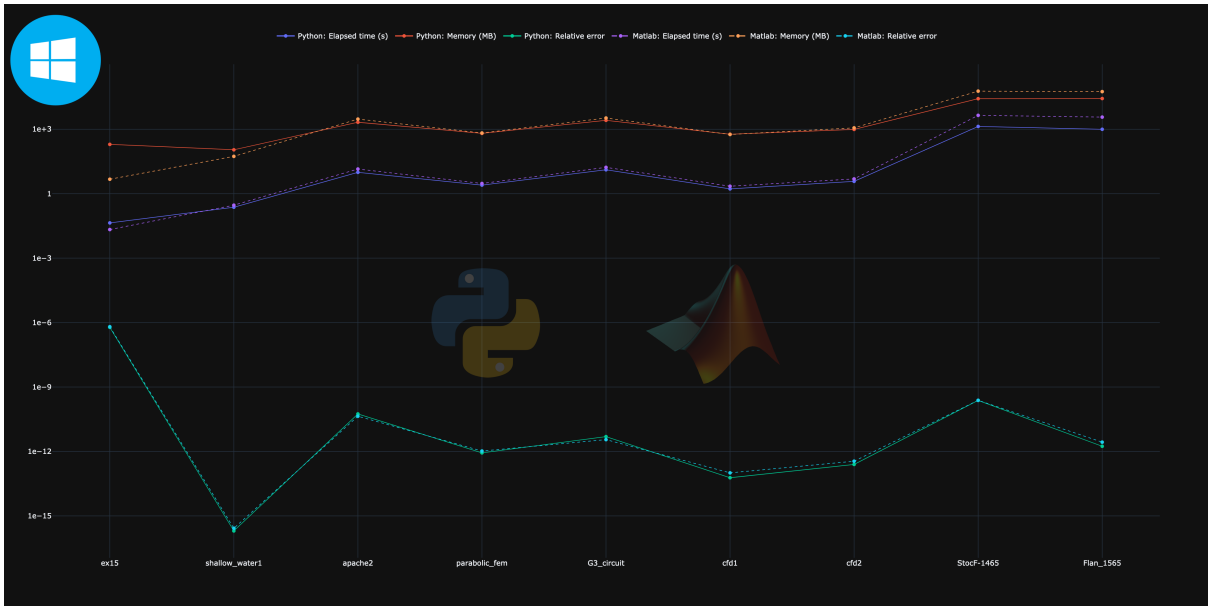

Figura 2: Confronto tra Python e MATLAB in Windows

5.1.2 Linux

Nella tabella 2 sono evidenziati i risultati ottenuti su Linux.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Python	837	9952	6.79E-08
MATLAB	3438	13815	7.39E-08

Tabella 2: Tempo, memoria ed errore medi su Linux

I risultati evidenziano, rispetto al caso di studio precedente, un netto miglioramento delle performance da parte di Python. Quest'ultimo è tre volte più veloce rispetto a MATLAB ed occupa anche meno memoria, una differenza di circa 4000 MebiByte come nel caso di Windows; l'errore relativo rimane immutato rispetto al caso precedente. Dal grafico 3 emerge come Python guadagni velocità al crescere dell'input e lo stesso comportamento è osservabile in termini di memoria; ciò sembrerebbe essere correlato alla percentuale di elementi diversi da zero salvati nella matrice, quindi dal livello di "sparsità" della matrice data in input, che potrebbe portare ad una differente gestione in termini di ottimizzazione da parte delle librerie coinvolte.

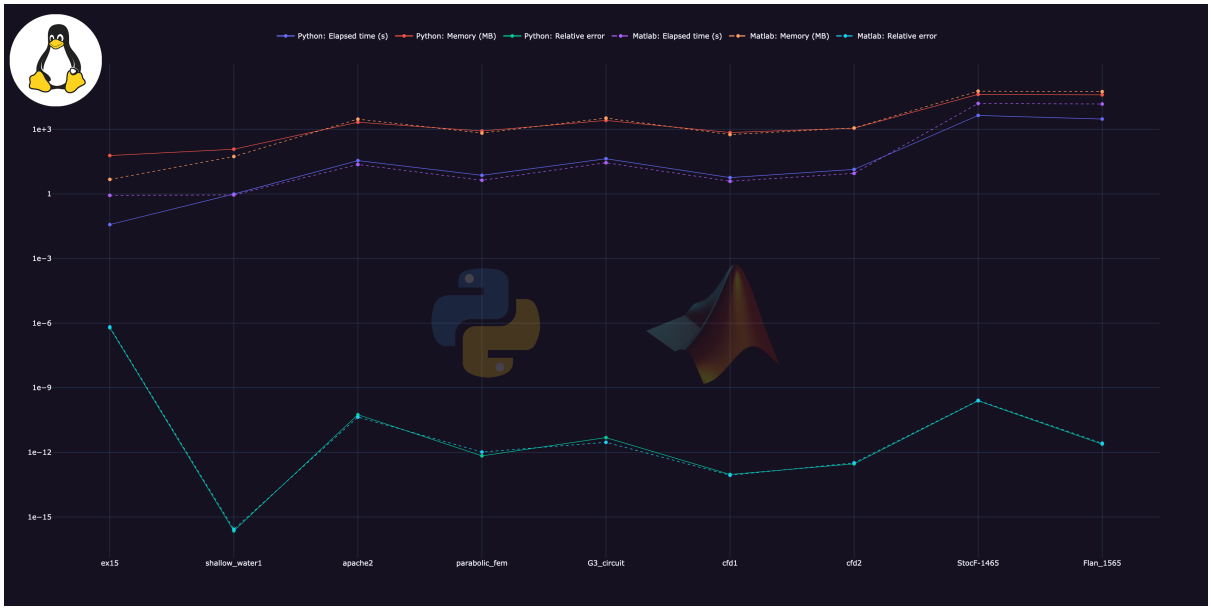

Figura 3: Confronto tra Python e MATLAB in Linux

5.1.3 MATLAB : confronto Linux - Windows

Nella seguente tabella 3 è riportato un confronto del software MATLAB nei diversi ambienti Linux e Windows.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Windows	1142	13815	7.39E-08
Linux	3438	13815	7.39E-08

Tabella 3: Tempo, memoria ed errore medi su MATLAB

Memoria ed errore relativo risultano uguali, coerentemente con la rappresentazione floating point utilizzata dall'ambiente MATLAB, per quanto riguarda la velocità di computazione su Windows, si ha un netto miglioramento delle performance, circa tre volte più veloce; questo comportamento è probabilmente dovuto alla diversa gestione dei processi (e presumibilmente anche della gestione dell'architettura) da parte dei sistemi coinvolti, che porterebbe a pensare che in ambiente Linux sia presente un overhead maggiore rispetto che in Windows, almeno per quanto riguarda i calcoli coinvolti. Il grafico 4 conferma l'analisi dei risultati e riporta la coerenza tra le computazioni effettuate sulle differenti matrici.

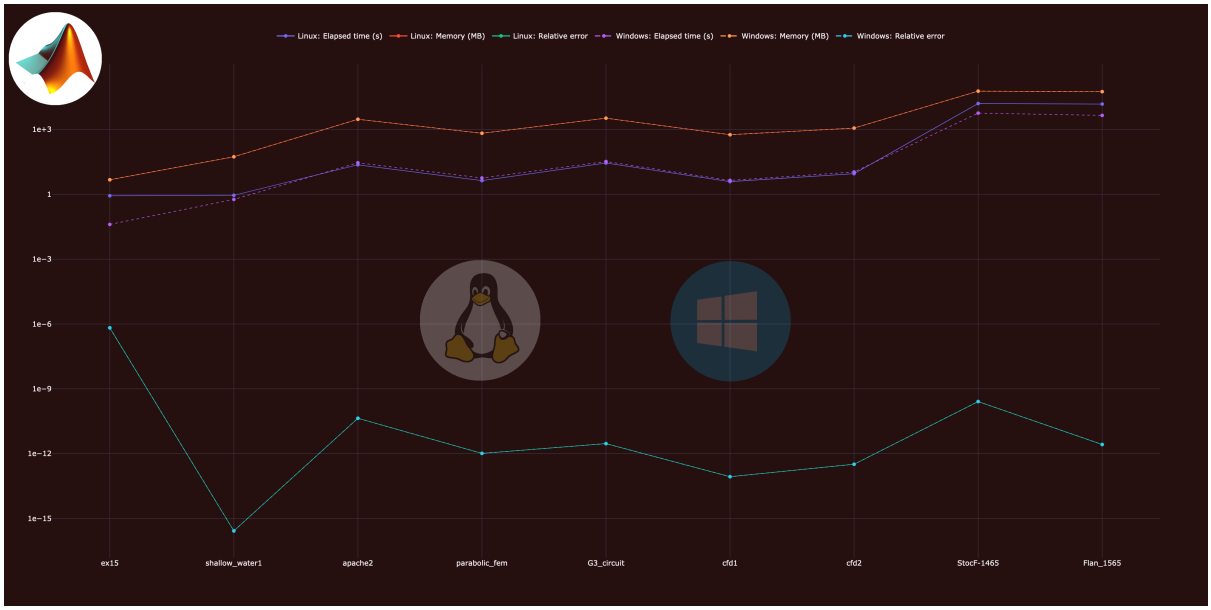

Figura 4: Confronto tra Windows e Linux in ambiente MATLAB

5.1.4 Python: confronto Linux - Windows

Nella tabella 4 viene riportato il confronto di Python tra i sistemi operativi Windows e Linux.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Windows	554	9232	6.79E-08
Linux	837	9952	6.79E-08

Tabella 4: Tempo, memoria ed errore medi su Python

I dati in questo caso non presentano sostanziali differenze, questo indica che il sistema operativo non influisce particolarmente sul processo computazionale in ambiente Python. Si noti come l'errore relativo non vari tra i due ambienti; tale nota sembrerebbe derivare sia dall'architettura hardware sottostante (che è la medesima), sia dall'interprete Python considerato (i due interpreti sono differenti in quanto compilati per i relativi sistemi operativi; la versione è stata mantenuta invariata per non introdurre ulteriori variabili e mantenere integrità nei dati raccolti). La costanza durante l'esecuzione è ulteriormente osservabile nel grafico 5 di seguito che rende visivamente molto difficile differenziare i due sistemi operativi.

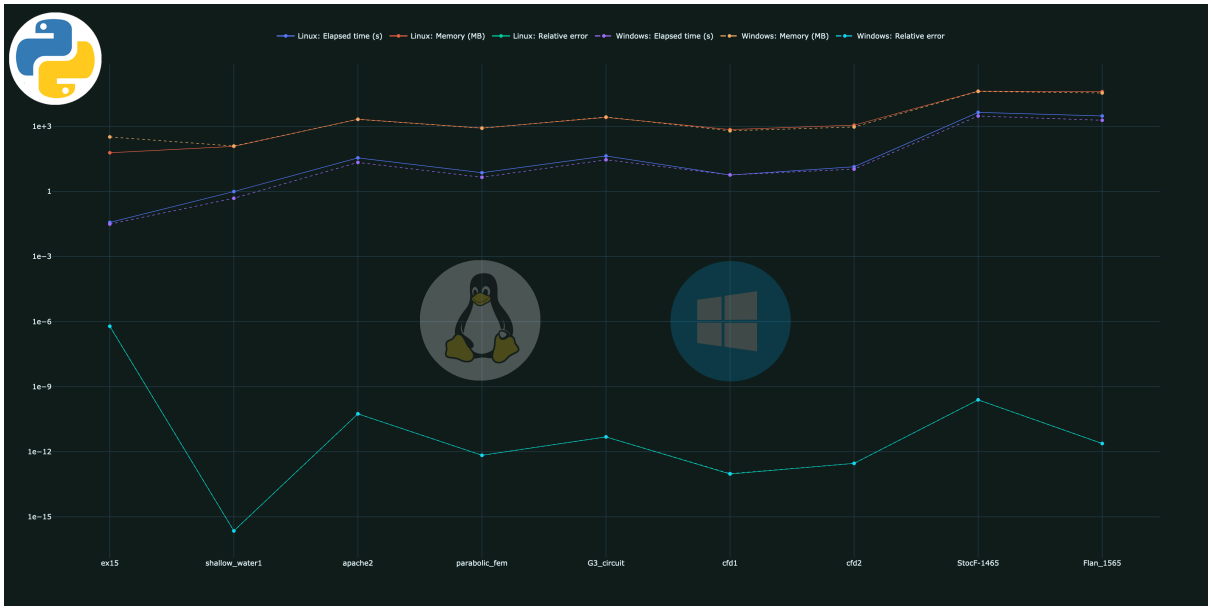

Figura 5: Confronto tra Windows e Linux in ambiente Python

5.2 Macchine aggiuntive

In questa sezione vengono rapidamente presentati due ulteriori casi di studio, interessanti per analizzare il comportamento di MATLAB e Python su una macchina differente (virtual machine) ed in un diverso ambiente (MacOS).

5.2.1 Virtual machine

La scelta di una virtual machine ci consente di confrontare il comportamento di Linux e Windows su una macchina con hardware differente dotato di una maggiore potenza computazionale. I dati di Windows sono riportati nella tabella 5.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Python	264	6718	6.69E-08
MATLAB	903	13815	7.15E-08

Tabella 5: Tempo, memoria ed errore medi su Windows

I dati confermano i risultati osservati sulla macchina di riferimento, cioè una migliore performance di Python rispetto a MATLAB; una peculiarità di quest'ultimo è la costanza nell'occupazione di memoria, che rimane invariata in ogni ambiente; ciò è dettato dalle politiche di gestione degli oggetti in MATLAB e dalla chiusura dell'ambiente stesso che permette di lavorare in una "sandbox" separata rispetto all'ambiente host. Nel grafico 6 viene riportata, anche in questo caso, l'analisi dei dati considerando ogni singola matrice. Anche in questo caso si riesce ad osservare una miglioria delle performance di Python al crescere dell'input.

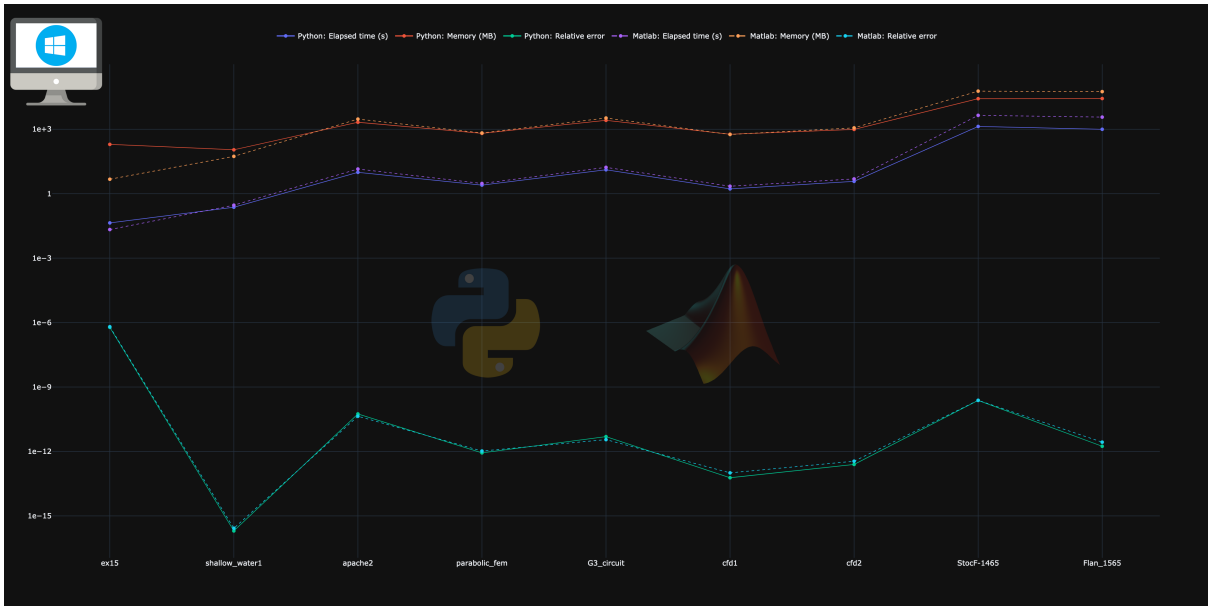

Figura 6: Confronto tra Python e MATLAB nella virtual machine con Windows

Sono stati effettuati i medesimi test in ambiente Linux, riportati nella tabella 6.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Python	420	9368	7.36E-08
MATLAB	1499	13815	7.15E-08

Tabella 6: Tempo, memoria ed errore medi su Linux

Anche in questo caso è facilmente visibile la migliore efficienza di Python rispetto alla controparte MATLAB.

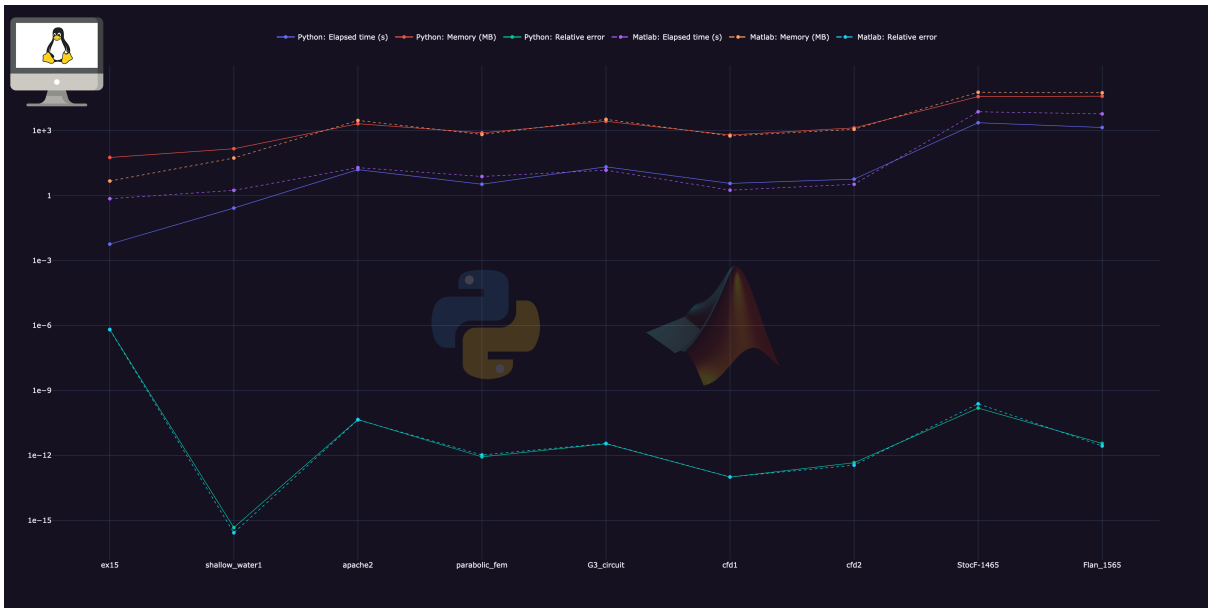

Figura 7: Confronto tra Python e MATLAB nella virtual machine con Linux

Il grafico 7 consente di confermare i risultati osservati (si noti che tali valori si discostano di poco per una questione di scala scelta, che pone enfasi sul fatto che non sono presenti differenze di ordini di grandezza in termini di metrica considerata).

5.2.2 MacOS - Darwin

È stato scelto di adottare un'ulteriore macchina per verificare le esecuzioni dei calcoli, così da avere una visuale completa per quanto riguarda i sistemi operativi più diffusi. Questo per verificare eventuali somiglianze o differenze evidenti rispetto ai precedenti test e per evidenziare eventuali anomalie riscontrate nelle differenti gestioni di risorse. In tabella 7 sono osservabili i risultati ottenuti in ambiente MacOS.

	Tempo (secondi)	Memoria (MiB)	Errore relativo
Python	137	7541	6.03E-08
MATLAB	496	13815	7.59E-08

Tabella 7: Tempo, memoria ed errore medi su Darwin

I dati evidenziano anche in questo caso la maggiore efficienza di Python, tuttavia è molto interessante osservare come MATLAB mantenga una costanza significativa di occupazione di memoria: in tutte le esecuzioni la memoria allocata rimane costante. Nel grafico 8 sono evidenziate le esecuzioni dei software, che risultano in linea con il resto dei test eseguiti.

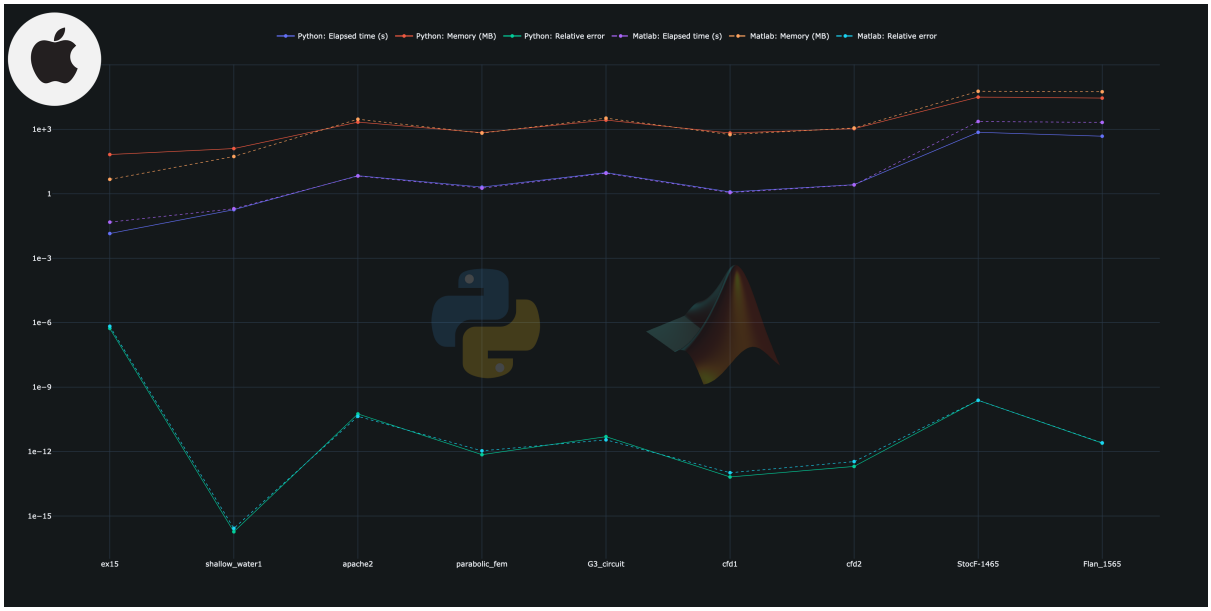

Figura 8: Confronto tra Python e MATLAB in Darwin

5.3 Considerazioni: memoria utilizzata

In questa sezione vengono effettuate delle brevi considerazioni sulla memoria utilizzata da Python per computare le matrici più grandi: StocF-1465 e Flan_1565. I grafici presi in considerazione (generati da uno script ad hoc implementato per evitare anomalie di misurazione date dall'utilizzo della memoria swap) mostrano la crescita della memoria utilizzata (ascisse: MebiByte) rispetto al tempo trascorso durante la computazione (ordinate: secondi). Si noti che è stato utilizzato un intervallo di campionamento pari a dieci secondi. È necessario porre molta attenzione al fatto che esistano una serie di incognite, che agiscono su tali dati: la differenza di gestione della ram su diversi sistemi operativi, eventuale compressione (in RAM) degli oggetti e dimensione della RAM stessa come componente hardware. La produzione di questi grafici è risultata utile per poter fornire un'idea pratica di come il carico di lavoro venga distribuito in base all'ambiente.

5.3.1 StocF-1465

I grafici 9 e 10 evidenziano una diversa gestione della memoria tra Windows e Linux. Nel primo caso vi è un'allocazione di memoria completa fin dal primo istante di esecuzione mentre nel secondo caso il sistema tende ad occupare la memoria man mano che viene richiesta, fino ad arrivare ad un picco; si hanno dunque due gestioni molto differenti anche dal punto di vista di allocazione temporale.

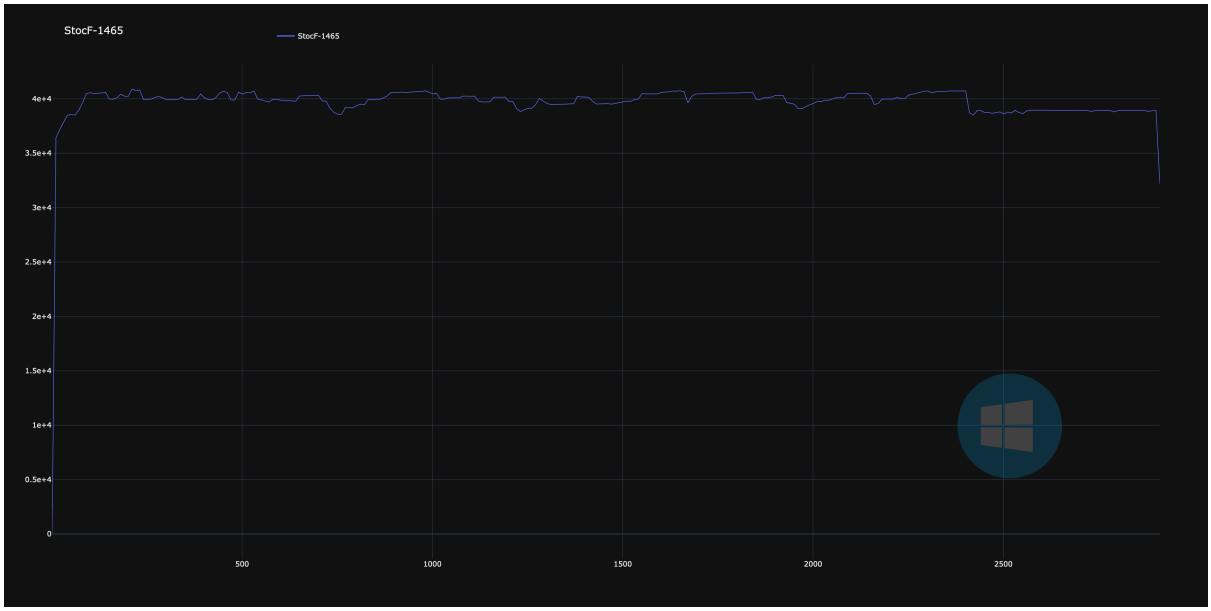

Figura 9: Utilizzo della memoria con StocF-1465 in Windows

Figura 10: Utilizzo della memoria con StocF-1465 in Linux

5.3.2 Flan_1565

Anche in questo caso, in figura 11 e 12 sono raffigurati i grafici di occupazione della memoria (per quanto riguarda la matrice Flan_1565). Il comportamento è analogo al caso precedente.

Figura 11: Utilizzo della memoria con Flan_1565 in Windows

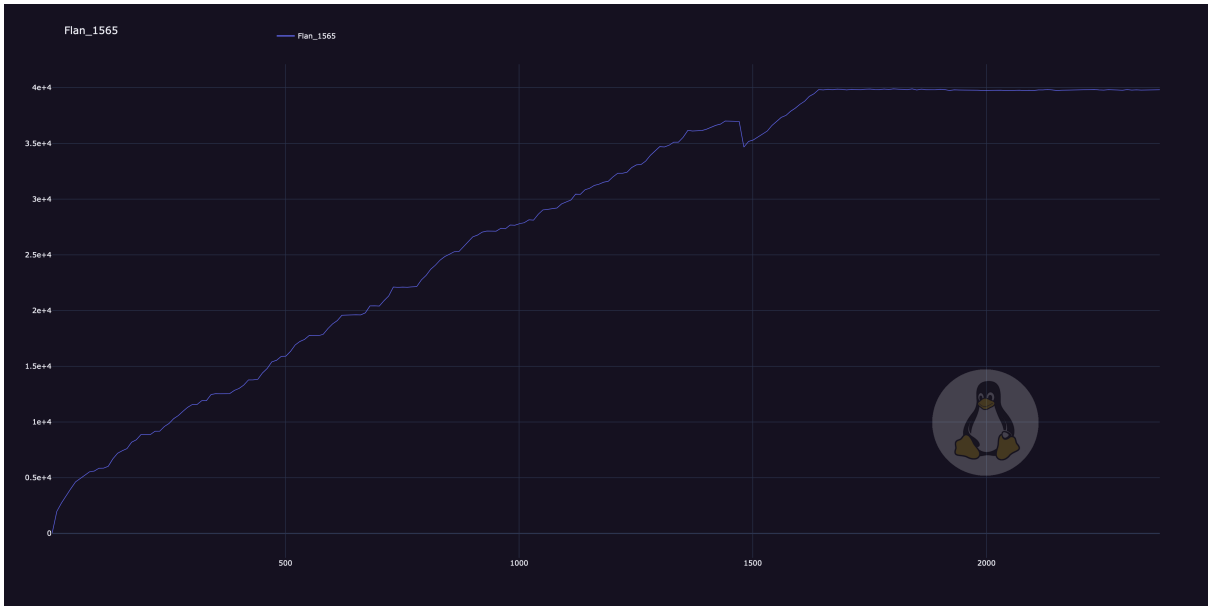

Figura 12: Utilizzo della memoria con Flan_1565 in Linux

5.3.3 Utilizzo della memoria in MacOS

Un ulteriore comportamento è evidenziato nei grafici 13 e 14, che riportano la memoria occupata da MacOS (Darwin) durante la computazione. Si può osservare un comportamento simile a Linux (coerentemente con il fatto che entrambi sono Unix-like), ma molto meno lineare e più altalenante (probabilmente per colpa del gestore della memoria che cerca di comprimere gli oggetti in RAM), con un picco principale verso la metà avanzata del calcolo.

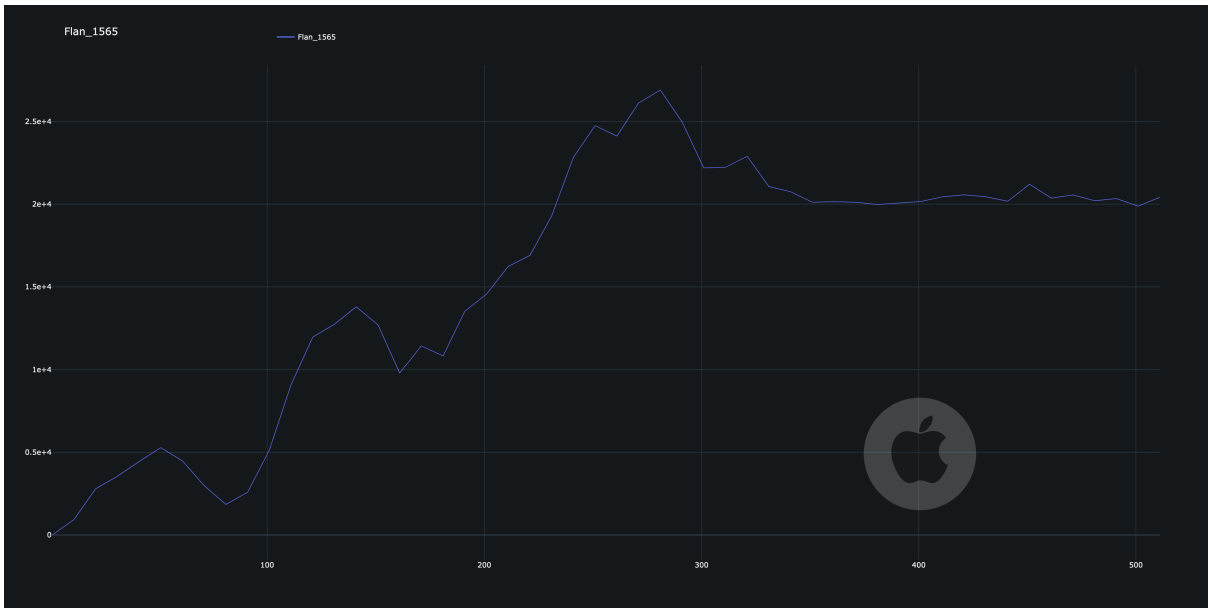

Figura 13: Utilizzo della memoria con Flan_1565 in Darwin

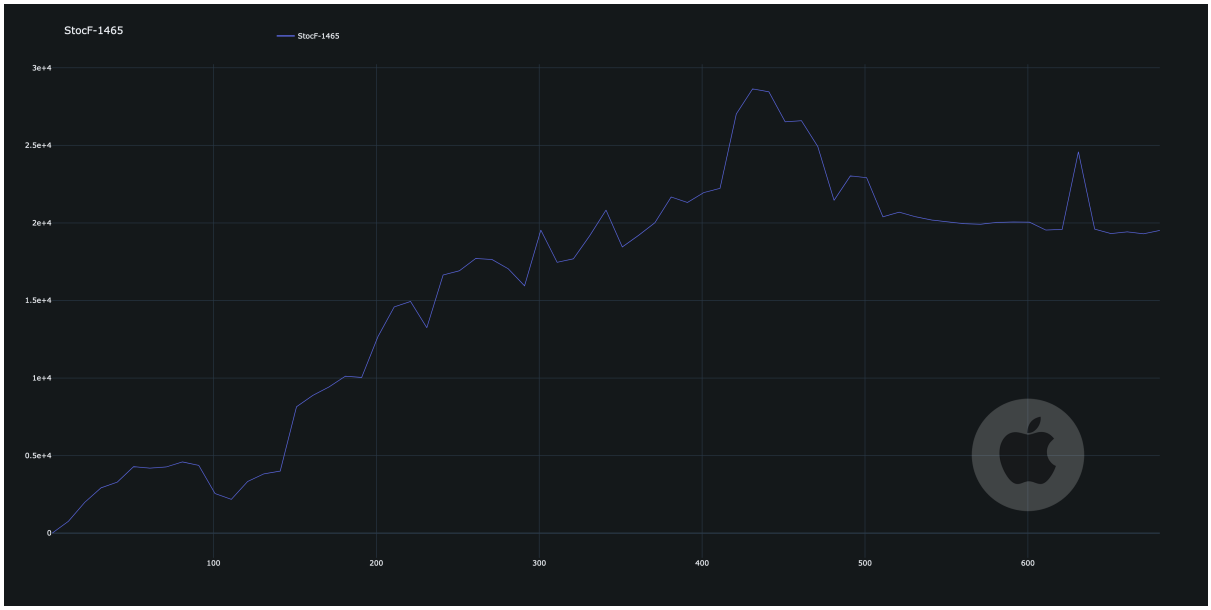

Figura 14: Utilizzo della memoria con StocF-1465 in Darwin

5.4 Codice

Di seguito viene illustrato il codice utilizzato per la creazione dei grafici mostrati precedentemente: lo script legge i dati presenti sui file csv e succeissivamente produce i grafici grazie alla libreria **plotly**.

```
1  import pandas as pd
2  import numpy as np
3  import plotly.graph_objects as go
4
5  # Global variables
6  INPUT_DIRECTORY = '/Users/'
7
8  def chart_os(os):
9      df_python = pd.read_csv(INPUT_DIRECTORY + 'data_python_' + os.lower
       () + '.csv')
10     df_matlab = pd.read_csv(INPUT_DIRECTORY + 'data_matlab_' + os.lower
       () + '.csv')
11
12     fig = go.Figure()
13
14     config = {'toImageButtonOptions': {'format': 'png', 'filename': '
       chart', 'height': 1000, 'width': 2000, 'scale': 2}}
15
16     fig.add_trace(go.Scatter(x=df_python['Matrix name'], y=df_python['
       Elapsed time (s)'], name='Python: Elapsed time (s)'))
17     fig.add_trace(go.Scatter(x=df_python['Matrix name'], y=df_python['
       Memory (MB)'], name='Python: Memory (MB)'))
18     fig.add_trace(go.Scatter(x=df_python['Matrix name'], y=df_python['
       Relative error'], name='Python: Relative error'))
19
20     fig.add_trace(go.Scatter(x=df_matlab['Matrix name'], y=df_matlab['
       Elapsed time (s)'], name='Matlab: Elapsed time (s)', line=dict(dash=
       'dash')))
21     fig.add_trace(go.Scatter(x=df_matlab['Matrix name'], y=df_matlab['
       Memory (MB)'], name='Matlab: Memory (MB)', line=dict(dash='dash')))
```

```
22      fig.add_trace(go.Scatter(x=df_matlab['Matrix name'], y=df_matlab['
        Relative error'], name='Matlab: Relative error', line=dict(dash='
        dash')))
23
24      fig.update_layout(title=os, showlegend=True, yaxis_type='log',
        yaxis=dict(showexponent='all', exponentformat='e'), template='
        plotly_dark', legend=dict(x=0.2, y=1.075), legend_orientation='h')
25      fig.show(config=config)
26
27  def chart_env(env):
28      df_linux = pd.read_csv(INPUT_DIRECTORY + 'data_' + env.lower() + '
        _linux.csv')
29      df_windows = pd.read_csv(INPUT_DIRECTORY + 'data_' + env.lower() +
        '_windows.csv')
30
31      fig = go.Figure()
32
33      config = {'toImageButtonOptions': {'format': 'png', 'filename': '
        chart', 'height': 1000, 'width': 2000, 'scale': 2}}
34
35      fig.add_trace(go.Scatter(x=df_linux['Matrix name'], y=df_linux['
        Elapsed time (s)'], name='Linux: Elapsed time (s)'))
36      fig.add_trace(go.Scatter(x=df_linux['Matrix name'], y=df_linux['
        Memory (MB)'], name='Linux: Memory (MB)'))
37      fig.add_trace(go.Scatter(x=df_linux['Matrix name'], y=df_linux['
        Relative error'], name='Linux: Relative error'))
38
39      fig.add_trace(go.Scatter(x=df_windows['Matrix name'], y=df_windows[
        'Elapsed time (s)'], name='Windows: Elapsed time (s)', line=dict(
        dash='dash')))
40      fig.add_trace(go.Scatter(x=df_windows['Matrix name'], y=df_windows[
        'Memory (MB)'], name='Windows: Memory (MB)', line=dict(dash='dash'))
        )
41      fig.add_trace(go.Scatter(x=df_windows['Matrix name'], y=df_windows[
        'Relative error'], name='Windows: Relative error', line=dict(dash='
        dash')))
42
43      fig.update_layout(title=env, showlegend=True, yaxis_type='log',
        yaxis=dict(showexponent='all', exponentformat='e'), template='
        plotly_dark', legend=dict(x=0.2, y=1.075), legend_orientation='h')
44      fig.show(config=config)
45
46  def chart_mem(mat, os):
47      dt_mem = np.loadtxt(INPUT_DIRECTORY + mat + '_' + os.lower() + '.
        txt', delimiter=',')
48
49      fig = go.Figure()
50
51      config = {'toImageButtonOptions': {'format': 'png', 'filename': '
        chart', 'height': 1000, 'width': 2000, 'scale': 2}}
52
53      fig.add_trace(go.Scatter(x=list(range(1, 10*(len(dt_mem)), 10)), y=
        dt_mem, name=mat))
54
55      fig.update_layout(title=mat, showlegend=True, yaxis=dict(
        showexponent='all', exponentformat='e'), template='plotly_dark',
        legend=dict(x=0.2, y=1.075), legend_orientation='h')
56      fig.show(config=config)
```

```
57
58 if __name__ == '__main__':
59     chart_os('Darwin')
60     chart_os('VM_Linux')
61     chart_os('VM_Windows')
62     chart_os('Linux')
63     chart_os('Windows')
64
65     chart_env('Python')
66     chart_env('Matlab')
67
68     chart_mem('StocF-1465', 'Darwin')
69     chart_mem('Flan_1565', 'Darwin')
70     chart_mem('StocF-1465', 'Linux')
71     chart_mem('Flan_1565', 'Linux')
72     chart_mem('StocF-1465', 'Windows')
73     chart_mem('Flan_1565', 'Windows')
```

Listing 4: charts.py

6 Conclusioni

Dopo aver raccolto i dati relativi all'implementazione del metodo di Cholesky e confrontato i risultati ottenuti, è dunque possibile effettuare una scelta motivata dal contesto di utilizzo (data-driven). Di seguito è riportato (Fig. 15) un riassunto dei dati raccolti categorizzati per ambiente, esposti in termini di medie e deviazioni standard.

			Windows	Linux	Darwin	VM Windows	VM Linux
Python	Media	Tempo (s)	554	837	137	264	420
		Memoria (MiB)	9232	9952	7541	6718	9368
		Errore Relativo	6.79E-08	6.79E-08	6.03E-08	6.69E-08	7.36E-08
	Deviazione Standard	Tempo (s)	1112	1669	273	522	849
		Memoria (MiB)	16255	17624	12880	11314	16405
		Errore Relativo	2.04E-07	2.04E-07	1.81E-07	2.01E-07	2.21E-07
MATLAB	Media	Tempo (s)	1142	3438	496	903	1499
		Memoria (MiB)	13815	13815	13815	13815	13815
		Errore Relativo	7.39E-08	7.39E-08	7.59E-08	7.15E-08	7.15E-08
	Deviazione Standard	Tempo (s)	2264	6807	979	1791	2983
		Memoria (MiB)	24972	24972	24972	24972	24972
		Errore Relativo	2.22E-07	2.22E-07	2.27E-07	2.14E-07	2.14E-07

Figura 15: Confronto completo tra i diversi ambienti

In entrambi gli ambienti è stato necessario intervenire sulle impostazioni di sistema per rendere compatibili le dimensioni dello spazio di swap su disco con le matrici da analizzare. In Windows, grazie alla quantità di documentazione ed alla semplicitá di configurazione, è risultato più rapido assegnare la giusta dimensione di swap. Per quanto riguarda la controparte Linux, invece, è stato necessario intervenire con metodologie meno user-friendly complicando inevitabilmente il processo. Un'osservazione importante riguarda il largo uso della memoria swap (i.e. operazioni di lettura e scrittura su disco rigido) che può comportare un peggioramento dello stato di salute del disco; è importante dunque valutare attentamente la scelta dei componenti di memoria e, per ogni ambiente, è importante valutare la quantità di memoria richiesta.

Utilizzando Windows è risultato complicato installare alcune librerie Python, difficoltá non riscontrata per Linux; tuttavia, quest'ultimo ha mostratato delle complicazioni durante l'installazione di MATLAB. Un ultimo appunto riguarda l'aspetto economico derivante dalla scelta del sistema operativo: Linux è gratuito ed Open Source, mentre Windows richiede una licenza di utilizzo.

Considerando i seguenti fattori:

- **Tempo di esecuzione:** è il maggior punto di forza di Python, indipendentemente dal sistema operativo, mediamente sempre più rapido di MATLAB.
- **Memoria:** Python risulta essere l'ambiente più efficiente, indipendentemente dal sistema operativo utilizzato, con una leggera ottimizzazione in Windows (può essere utile nel caso di grosse e continue elaborazioni). MATLAB occupa un quantitativo di memoria costante in tutti i test.
- **Errore relativo:** nessuna differenza sostanziale tra gli ambienti o i sistemi operativi; in contesti critici Python sembra comportarsi lievemente meglio.

A seguito di queste considerazioni, è importante introdurre ulteriori precisazioni che consentono di ottenere un adeguato compromesso:

- MATLAB è un software proprietario (con un costo di 2000 € per una singola licenza standard [29]). Python è gratuito ed open source.
- entrambi i linguaggi di programmazione sono sintatticamente e semanticamente ad alto livello.
- MATLAB dispone di supporto tecnico business, Python no.
- l'installazione delle librerie in ambiente Python aggiunge un livello di incertezza e complicazione rispetto a MATLAB, che gestisce in modo autonomo l'installazione/ rimozione di eventuali aggiornamenti.
- le librerie utilizzate in MATLAB risultano ben documentate e direttamente accessibili all'interno dell'ambiente. Quelle presenti per Python sono spesso aggiornate, facilmente estendibili/modificabili e complete.

Per concludere, rispondendo alla richiesta della traccia:

> "È meglio affidarsi alla sicurezza di MATLAB pagando oppure vale la pena di avventurarsi nel mondo open source? Ed è meglio lavorare in ambiente Linux oppure in ambiente Windows?"

Non esiste una risposta diretta. Quest'ultima è la conseguenza di considerazioni ed osservazioni che necessariamente derivano dal contesto aziendale di riferimento (devono comprendere anche aspetti secondari, ad esempio la formazione di risorse umane) e, più in generale, dal dominio di interesse.

7 Struttura della directory

- /Progetto 1/: root
  - /Progetto 1/**charts.py:** script in Python utilizzato per la realizzazione dei grafici di confronto
  - /Progetto 1/**cholesky_matlab.m:** programma MATLAB
  - /Progetto 1/**cholesky_python_memory.py:** script in Python di supporto per il calcolo della memoria impiegata dalle matrici che utilizzano la memoria swap in ambiente Python
  - /Progetto 1/**cholesky_python.py:** programma Python

Sitografia

[1] URL: https://sparse.tamu.edu/.

[2] URL: https://sparse.tamu.edu/Janna/Flan_1565.

[3] URL: https://sparse.tamu.edu/Janna/StocF-1465.

[4] URL: https://sparse.tamu.edu/Rothberg/cfd2.

[5] URL: https://sparse.tamu.edu/Rothberg/cfd1.

[6] URL: https://sparse.tamu.edu/AMD/G3_circuit.

[7] URL: https://sparse.tamu.edu/Wissgott/parabolic_fem.

[8] URL: https://sparse.tamu.edu/GHS_psdef/apache2.

[9] URL: https://sparse.tamu.edu/MaxPlanck/shallow_water1.

[10] URL: https://sparse.tamu.edu/FIDAP/ex15.

[11] URL: https://en.wikipedia.org/wiki/Just-in-time_compilation.

[12] URL: https://www.r-project.org/.

[13] URL: https://www.python.org/downloads/release/python-377/.

[14] URL: https://it.mathworks.com/login?uri=%5C%2Fdownloads%5C%2Fweb_downloads%5C%2Fdownload_release%5C%3Frelease%5C%3DR2020a.

[15] URL: https://azure.microsoft.com/it-it/.

[16] URL: https://en.wikipedia.org/wiki/Comma-separated_values.

[17] URL: https://pypi.org/project/psutil/.

[18] URL: https://pypi.org/project/memory-profiler/.

[19] URL: https://docs.scipy.org/doc/scipy/reference/linalg.html.

[20] URL: https://scikit-sparse.readthedocs.io/en/latest/.

[21] URL: https://github.com/xmlyqing00/Cholmod-Scikit-Sparse-Windows.

[22] URL: https://cvxopt.org/documentation/index.html.

[23] URL: https://developer.nvidia.com/cholmod.

[24] URL: https://github.com/DrTimothyAldenDavis/SuiteSparse/tree/master/CHOLMOD.

[25] URL: https://it.mathworks.com/help/matlab/ref/mldivide.html.

[26] URL: https://it.mathworks.com/help/matlab/ref/chol.html.

[27] URL: http://www.alecjacobson.com/weblog/?p=4161.

[28] URL: https://it.mathworks.com/matlabcentral/answers/357776-is-it-possible-to-speed-up-solving-ax-b-for-multiple-b-s-by-pre-computing-the-cholesky-factorization.

[29] URL: https://it.mathworks.com/pricing-licensing.html?prodcode=ML&intendeduse=comm.