Università degli Studi di Firenze

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in

INFORMATICA

Una libreria di algebra lineare per il calcolo scientifico

Tesi di laurea di

Sandro Tosi

Relatore:

prof. Luigi Brugnano

Anno accademico 2002/2003

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

In questo capitolo, si cercherà di mostrare come la risoluzione di sistemi lineari sia una pratica fondamentale in un ampio spettro di campi applicativi e quali possano essere le motivazioni per lo sviluppo di un metodo che faccia uso di una quantità ridotta di memoria rispetto ad altri strumenti.

1.1 Risoluzione di sistemi lineari e loro applicazione

Da sempre il progresso scientifico si basa sulla stretta alternanza tra l'osservazione di un fenomeno e la formulazione di una teoria o di un modello che lo descriva. Questo modello deve essere mantenuto il più semplice possibile, introducendo alcune assunzioni semplificatrici (per esempio, nello studio dei pianeti, essi vengono solitamente considerati come punti dotati di massa), ma tutti gli effetti rilevanti devono essere presi in considerazione. Quanto elaborato deve poi essere verificato sia sotto l'aspetto della correttezza (coerenza nella descrizione dell'evento), sia dal punto di vista della completezza (precisione della descrizione).

Il modello, quindi, è utilizzato per fare previsioni sull'evoluzione del fenomeno (si considerino le previsioni del tempo). Questo si concretizza con la sua simulazione su calcolatore.

Le simulazioni numeriche sono spesso assai onerose, dal punto di vista computazionale, in quanto la dimensione del problema da risolvere può essere elevata. Queste necessità portano alla richiesta di un'elevata potenza di calcolo: non è un caso, infatti, che i più grandi supercalcolatori siano stati costruiti proprio in risposta alle richieste del calcolo scientifico.

Un gran numero di problemi scientifici, si stima addirittura il 75% (cfr. [5, pg. 137]), necessita di dover risolvere un sistema di equazioni lineari. Questa attività è infatti uno di quei problemi "di base" che ricorrono molto di frequente nella pratica, e risulta essere un elemento standard nei calcoli numerici ed un costituente fondamentale per molte applicazioni scientifiche, dell'ingegneria, dell'industria ed in tutti quei campi in cui si fa uso di modelli matematici.

Nell'ambito della fisica si trovano molte applicazioni dirette alla risoluzione dei sistemi lineari, per esempio nello studio dei circuiti elettrici: il flusso di corrente in un circuito elettrico deve soddisfare le leggi di Kirchhoff (cfr. [11, pg. 773, 779]) per il bilanciamento delle differenze di potenziale all'interno dei cicli e delle correnti nei nodi. Queste equazioni sono lineari nei valori delle resistenze e delle sorgenti di energia e la risoluzione del sistema associato restituisce le correnti attraverso ogni nodo del circuito (cfr. [3] [4]).

Un altro esempio in cui l'applicazione dei sistemi lineari è diretta si ha nei problemi di statica in meccanica strutturale: si consideri il calcolo dello stress e del bilanciamento delle forze in una struttura come un edificio, il telaio di un aereo oppure un ponte; in quest'ultimo caso si ha che le forze dovute alla massa del ponte e dei mezzi che lo percorrono devono essere bilanciate dai pilastri su cui poggia, così come le forze di torsione in ogni giunzione devono essere nulle, altrimenti il ponte inizierebbe a oscillare. Questa evenienza si è talora verificata, come nel caso del Takoma National Bridge (cfr. [7] [21]). Scomponendo le forze nelle componenti verticali ed orizzontali si ottengono, in ogni nodo, i vincoli $\sum f_x = 0$ e $\sum f_y = 0$. Studiando il ponte nella sua interezza si giunge ad un sistema lineare di vincoli la cui risoluzione porta a conoscere le forze di sollecitazione nella struttura (cfr. [3] [14] [15]).

Anche se non in modo diretto, la risoluzione di sistemi lineari ricorre con frequenza come passaggio intermedio di un problema più complesso: la risoluzione di un sistema non lineare tramite il metodo di Newton, ad esempio, richiede di risolvere un sistema lineare ad ogni iterazione.

Un altro esempio riguarda la soluzione di equazioni differenziali, il cui utilizzo è fondamentale in ogni ambito scientifico: infatti, l'utilizzo dei cosiddetti metodi impliciti, genera una successione di sistemi non lineari da risolvere. Come detto innanzi, la risoluzione di ciascuno di questi richiede di risolvere un certo numero di sistemi lineari.

Anche in altre situazioni si richiede la risoluzione di sistemi lineari: problemi di vibrazioni, discretizzazione di problemi continui come integrali ed equazioni differenziali, approssimazioni ai minimi quadrati, modellizzazione di antenne, analisi di campi elettromagnetici, ecc.

Diverse sono anche le applicazioni in ambiti molto vari come per le previsioni del tempo, le perforazioni petrolifere, il calcolo delle traiettorie dei satelliti, gli studi medici ed i modelli per la biologia, l'analisi di reazioni chimiche, la crittografia e lo studio dei cambiamenti climatici.

Non si deve, però, pensare che se ne faccia uso solo in campi strettamente scientifici: si trovano applicazioni nelle scienze sociali, per i modelli comportamentali, ed in economia, dove sono stati sviluppati complessi modelli matematici per il bilancio dei flussi di denaro.

Più in generale, tutti i problemi le cui relazioni sono lineari, possono richiedere la risoluzione di sistemi lineari.

1.2 Motivazioni

Sistemi lineari di grandi dimensioni possono presentare problemi di occupazione di memoria, e la necessità di dover risolvere questi sistemi, ha portato l'interesse verso metodi alternativi che guardassero maggiormente al risparmio nell'uso della memoria, piuttosto che alla rapidità di esecuzione.

Lo scopo del presente lavoro di tesi è lo sviluppo di due librerie di software matematico (una in Fortran ed una in C) per la risoluzione efficiente, riguardo alla occupazione di memoria, di sistemi lineari densi di equazioni lineari.

Capitolo 2

Il metodo di Wassing

In questo capitolo verrà presentato il metodo di Wassing [23] [12], che consente la risoluzione di un sistema lineare con un utilizzo di memoria quattro volte inferiore rispetto all'eliminazione di Gauss. Questo è possibile grazie al fatto di non richiedere che la matrice dei coefficienti sia disponibile interamente in memoria, ma che sia possibile ottenerne gli elementi solo quando necessario.

2.1 Basi teoriche del metodo di Wassing

Il nostro problema è quello di risolvere un sistema lineare

$$Ax = b$$

con A matrice $n \times n$ non singolare e densa.

Un classico modo per risolvere questo problema, consiste nel fattorizzare LU la matrice A:

$$A = LU,$$

con L matrice triangolare inferiore a diagonale unitaria, U matrice triangolare superiore e risolvere, quindi, i sistemi triangolari

$$Ly = b, \quad Ux = y.$$

Questo ha un costo di n^2 locazioni di memoria e circa $\frac{2}{3}n^3$ flops[1]. Di questo metodo, esiste la nota variante con pivoting parziale, largamente utilizzata nella pratica computazionale, ed implementata in numerose librerie di software matematico.

Per il metodo di Wassing, si costruisce, a partire dal sistema lineare $Ax = b$, una nuova matrice $(n+1) \times n$, chiamata Z, nel seguente modo:

$$Z = \left[\begin{array}{c} A^T \\ \hline -b^T \end{array}\right].$$

Se, dunque, si fattorizza la matrice Z come LU, si ottiene

$$Z = \left[\begin{array}{c} A^T \\ \hline -b^T \end{array}\right] = \left[\begin{array}{c|c} L & 0 \\ \hline l^T & 1 \end{array}\right] \cdot \left[\begin{array}{c} U \\ \hline \underline{0}^T \end{array}\right] \equiv \hat{L}\hat{U}$$

dove $L, U \in \mathbb{R}^{n \times n}$ e $\underline{0}, l \in \mathbb{R}^n$.

Da questo, eseguendo formalmente il prodotto tra le due matrici, risulta che

$$\left[\begin{array}{c} A^T \\ \hline -b^T \end{array}\right] = \left[\begin{array}{c} LU \\ \hline l^T U \end{array}\right],$$

ovvero

$$-b^T = l^T U \quad \text{e} \quad A^T = LU.$$

Pertanto,

$$b = -U^T l \quad \text{e} \quad A = U^T L^T.$$

[1] Il termine *flop* è l'acronimo di *fl*oating-point *op*eration.

Ricordando che il problema di partenza era $Ax = b$, si vede come sia possibile ricondurlo a questa forma: $U^T L^T x = -U^T l$ e cioè

$$L^T x = -l \quad \Rightarrow \quad x^T = -l^T L^{-1}.$$

Si analizzi ora più in dettaglio la matrice $\hat{L}$: essa ha forma

$$\hat{L} = \left[\begin{array}{c|c} L & 0 \\ \hline l^T & 1 \end{array} \right].$$

Considerandone l'inversa,

$$\hat{L}^{-1} = \left[\begin{array}{c|c} L^{-1} & 0 \\ \hline -l^T L^{-1} & 1 \end{array} \right],$$

si osserva che i primi n elementi dell'ultima riga coincidono con il vettore soluzione x^T.

Pertanto, si richiede di dover calcolare soltanto $\hat{L}$ (e non anche U) per ottenere la soluzione al sistema lineare di partenza.

2.2 Descrizione dell'algoritmo

Sia $Z = (z_{ij}) \in \mathbb{R}^{n+1 \times n}$, allora i passi dell'algoritmo di Wassing sono i seguenti:

1. $\forall i = 2, 3, \ldots, n, n+1$

   $z_{i1} = -z_{i1}/z_{11}$

2. $\forall k = 2, 3, \ldots, n,$

   (a) $\forall i = k, k+1, \ldots, n, n+1$
   $$z_{ik} = z_{ik} + \sum_{j=1}^{k-1} z_{ij} z_{jk}$$

   (b) $\forall i = k+1, k+2, \ldots, n, n+1$

   $z_{ik} = -z_{ik}/z_{kk}$

   (c) $\forall j = 1, 2, \ldots, k-1$ e $\forall i = k+1, k+2, \ldots, n, n+1$

   $z_{ij} = z_{ij} + z_{ik}z_{kj}$

3. Il vettore soluzione, sarà

   $x_j = z_{n+1,j}\ \forall j = 1, 2, \ldots, n$

2.3 Ottenere un'occupazione di memoria di $(n+1)^2/4$ elementi

Come detto in precedenza, per la risoluzione del sistema lineare si ha bisogno soltanto della matrice $\hat{L}$: dal momento che questa è una matrice triangolare inferiore a diagonale unitaria, se si memorizzasse interamente si avrebbe un'occupazione di memoria di approssimativamente $n^2/2$ elementi.

È possibile, tuttavia, ottenere un'occupazione ancora minore di memoria, eliminando gli elementi che, nei passi successivi, non verranno più utilizzati. Infatti, dalla descrizione fatta in precedenza, si può notare che, alla fine del passo 1, si ha bisogno di memorizzare n elementi (quelli sottodiagonali della prima colonna: $[z_{21} \ldots z_{n+1,1}]$); in seguito, al termine del secondo passo, si devono memorizzare $2(n-1)$ elementi (gli elementi sottodiagonali della seconda colonna, $[z_{32} \ldots z_{n+1,2}]$, e quelli sulle stesse righe della prima colonna, $[z_{31} \ldots z_{n+1,1}]$). In generale, alla fine del passo i-esimo solo gli elementi sulle righe $i+1, \ldots, n+1$ delle prime i colonne saranno necessari per il passo successivo. Pertanto, si ottiene la seguente relazione per l'occupazione di memoria ai vari dei passi:

$$f(i) = i(n - i + 1), \quad i = 1, \dots, n.$$

Si vede facilmente che $f(i)$ raggiunge il valore massimo in corrispondenza di $i = \frac{n+1}{2}$ (assumendo, per semplicità, che tale quantità sia intera): pertanto, la massima occupazione di memoria richiesta sarà:

$$f\left(\frac{n+1}{2}\right) = \frac{(n+1)^2}{4}.$$

È dunque possibile ottenere un notevole risparmio di occupazione di memoria, a patto però di non utilizzare esplicitamente una matrice per contenere gli elementi intermedi della risoluzione (altrimenti verrebbero usati circa $(n+1)^2$ elementi) ma un vettore, in cui l'accesso sia svolto in maniera opportuna.

Capitolo 3

Implementazione

Si discuteranno qui alcune scelte implementative ed alcuni accorgimenti adottati nell'implementazione del metodo di Wassing.

3.1 Schema di accesso e modifica di `z`

Come descritto in precedenza la matrice Z è memorizzata in un vettore, chiamato `z`, il quale contiene esclusivamente gli elementi di Z necessari ai passi successivi. Questi elementi sono memorizzati per colonne:

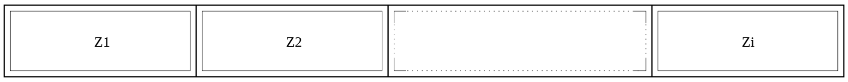

Figura 3.1: Rappresentazione dell'allocazione di memoria di `z`

dove con Z_i si è indicata l'i-esima colonna di Z.

Gli elementi in questo vettore sono letti e successivametne modificati: al passo i-esimo si ha bisogno di accedere agli elementi della sotto-matrice di Z di indici $(i \ldots n+1, 1 \ldots i-1)$ che sono già presenti in `z`. Alla fine di questo passo la riga i-esima non è più necessaria: si deve quindi eliminare il primo elemento di tutte le sotto-colonne memorizzate in `z`, compattando gli elementi rimasti, ed in più si devono memorizzare gli elementi $i+1 \ldots n+1$ della colonna i.

Un esempio renderà più chiaro lo schema di allocazione[1]: si supponga, per semplicità, che la matrice A sia 3×3, quindi la matrice Z ha dimensioni 4×3 ed il vettore deve essere dimensionato per contenere $(n+1)^2/4 = 4^2/4 = 4$ elementi.

Al primo passo non si ha bisogno di elementi memorizzati in `z`, ma si dovranno solo memorizzare gli elementi della colonna corrente necessari in seguito:

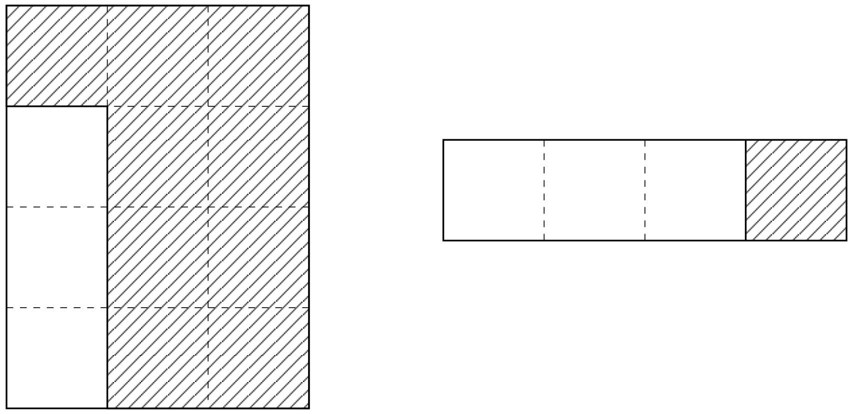

Figura 3.2: Contenuto di Z e `z` al termine del passo 1

Il passo 2 necessita degli elementi appena memorizzati per aggiornare

[1]Oltre alla descrizione delle operazioni svolte, verranno graficati il contenuto della matrice Z e del vettore `z`: per Z, gli spazi "righettati" sono le aree di memoria non utilizzate e quelli barrati sono gli spazi che erano stati utilizzati ma che ora non sono più necessari; per `z`, le aree "righettate" sono quelle non utilizzate al passo corrente ed il cui contenuto non ci interessa.

la colonna corrente; questa verrà modificata ed si memorizzano gli elementi utili:

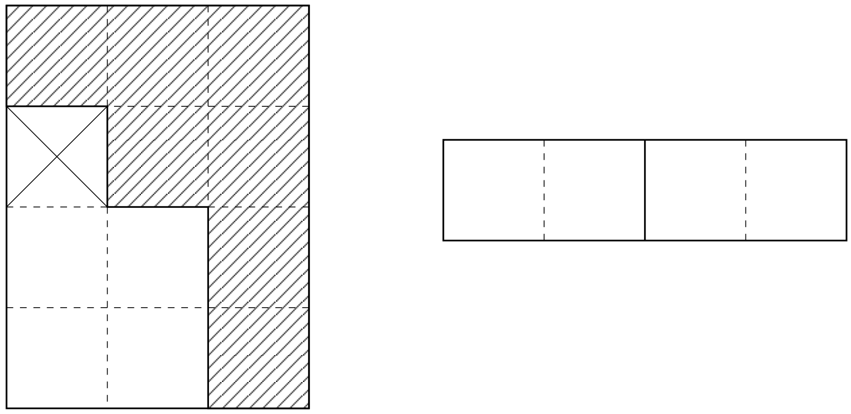

Figura 3.3: Contenuto di Z e $\mathtt{z}$ al termine del passo 2

Infine, alla terza ed ultima iterazione, dopo l'aggiornamento della colonna corrente e la memorizzazione degli elementi opportuni si avrà la seguente situazione:

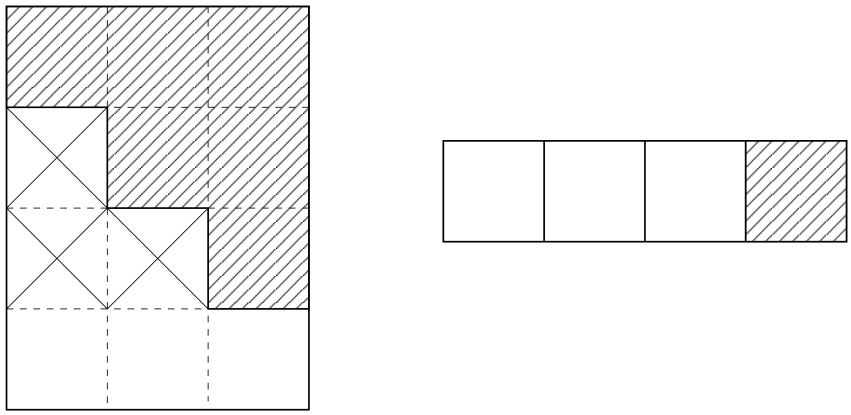

Figura 3.4: Contenuto di Z e $\mathtt{z}$ al termine del passo 3

dove l'ultima riga di Z e i primi n elementi di $\mathtt{z}$ contengono la soluzione cercata.

Un esempio maggiormente complesso è dato in figura 3.5 (di cui si mostrerà solo il contenuto del vettore z durante i vari passi):

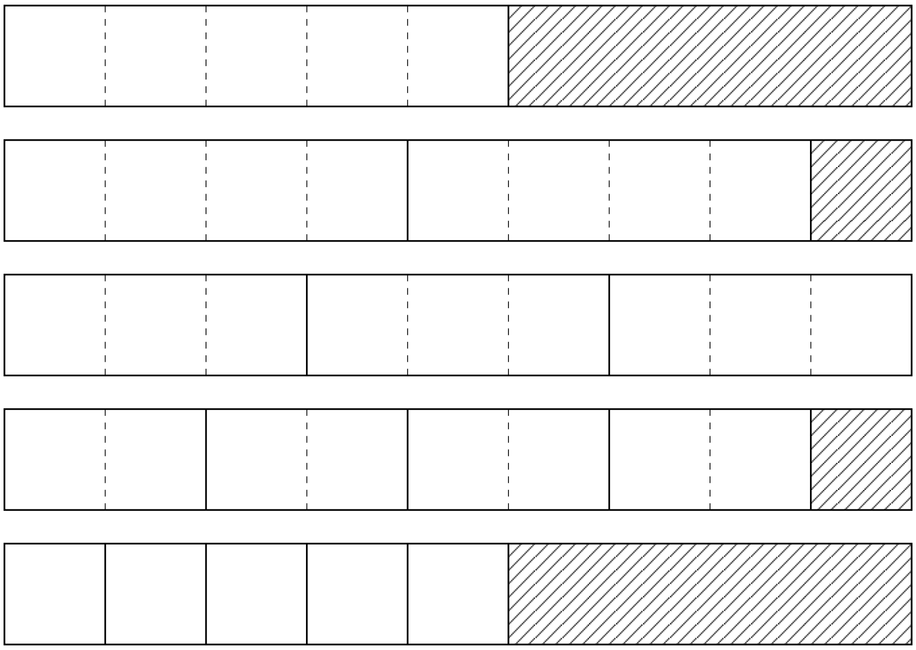

Figura 3.5: Contenuto di z per una matrice A 5×5

Per accedere agli elementi di z si è usato il seguente schema: si supponga di essere al passo k e di dover accedere all'elemento di indici (i, j) (con $i \geq k$ e $j < k$, per quanto visto in precedenza); si è visto che z contiene gli elementi di Z ordinati per colonne, ognuna delle quali contiene $n-k+2$ elementi: dal momento che per accedere alla prima colonna non si ha bisogno di un *offset*, in quanto questa si trova all'inizio di z, per ottenere l'indice dell'elemento precedente l'inizio della colonna j, si calcola $\text{offset}_j = (n-k+2)(j-1)$. Ora, per accedere all'elemento $(k+1, j)$ si deve incrementare di 1 offset_j ed allora per accedere all'elemento (i, j) si deve aggiungere la quantità $i - k + 1$. In conclusione:

$$Z[i,j] \;=\; \mathtt{z}[\underbrace{(n-k+2)(j-1)}_{\text{offset}_j} + \underbrace{i-k+1}_{(i,j)}]$$

3.2 Il caso di più termini noti

Frequentemente si ha la necessità di risolvere sistemi lineari in cui la matrice dei coefficienti rimane costante, ma dove il vettore dei termini noti cambia: è quindi di grande interesse poter risolvere in modo efficiente più sistemi lineari con la stessa matrice dei coefficienti ma con differenti termini noti, eseguendo la fattorizzazione della matrice solamente una volta.

Il problema, adesso, è il seguente: dato il sistema

$$AX = B$$

con $A \in \mathbb{R}^{n\times n}$ e $X, B \in \mathbb{R}^{n\times m}$, determinare la matrice X che lo soddisfa.

Le colonne della matrice soluzione X, $x^{(i)}$, sono i vettori soluzione degli m sistemi lineari

$$Ax^{(i)} = b^{(i)} \qquad i = 1, \ldots, m,$$

uno per ogni vettore colonna $b^{(i)}$ della matrice dei termini noti.

L'applicazione del metodo di Wassing, in questo caso, cambia solo nella sua organizzazione. Idealmente, è come se la matrice Z adesso fosse strutturata in:

$$Z = \left[\begin{array}{c} A^T \\ \hline -B^T \end{array}\right]$$

dove, al posto del precedente vettore $-b^T$, si ha la matrice $-B^T$.

In realtà, le matrici A e B sono mantenute separate; la prima nel vettore `z`, come mostrato in precedenza, la seconda in un altro vettore, `b`.

Siano m i vettori $b^{(i)}$, allora la matrice B viene riorganizzata nel vettore `b` come mostrato in figura 3.6: siano, cioè, B_i, $i = 1, \ldots, n$ le n righe di B, allora esse saranno poste l'una di seguito l'altra nel vettore `b`.

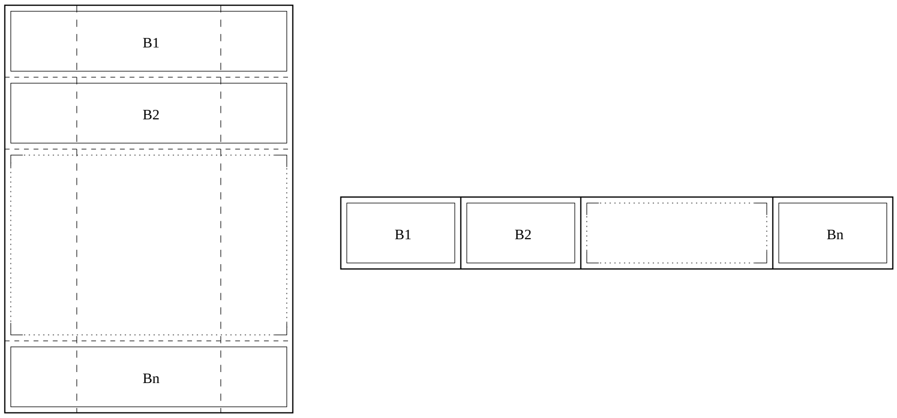

Figura 3.6: Rappresentazione dell'allocazione di memoria del vettore `b`

Utilizzando questo metodo di allocazione, le operazioni di aggiornamento avvengono in maniera efficiente; gli elementi da modificare si trovano sulla stessa riga della matrice B: memorizzando in modo consecutivo questi elementi nel vettore `b`, l'accesso ad essi avverrà in maniera sequenziale, evitando perciò accessi non contigui in memoria.

Dal momento che l'algoritmo per più termini noti è più generale, le librerie implementate dispongono soltanto di questa versione. Per questo, il vettore `z` dovrà essere dimensionato per contenere $n^2/4$ elementi, mentre il vettore `b` dovrà contenere mn elementi.

3.3 Gestione della memoria

La gestione della memoria, ed in particolare dell'allocazione dei vettori, avviene con modalità differenti tra il Fortran ed il C.

Il Fortran prevede che i vettori necessari all'esecuzione delle subroutine siano dichiarati prima nel programma principale, per poi essere passati, come parametri, ai sottoprogrammi.

In C, invece, è possibile far uso di funzioni di allocazione dinamica della memoria per la dichiarazione dei vettori direttamente all'interno delle subroutine, senza percui aver bisogno di parametri aggiuntivi.

3.4 Implementazione del pivoting parziale

La tecnica del pivoting parziale consiste nello scambiare le righe della matrice in modo da posizionare un elemento particolarmente desiderabile nella posizione diagonale, dalla quale l'elemento pivot deve essere scelto.

Si è optato, allora, per usare la metodologia classica di scelta del pivot come l'elemento di modulo massimo nella colonna corrente. È noto, infatti, che questa strategia consente di ottenere un metodo numerico stabile.

Per l'applicazione del pivoting parziale vengono mantenuti due vettori (inizialmente contengono $\texttt{vect}[i] = i$):

`ip`: contiene la permutazione delle righe della matrice fino al passo corrente; alla fine della risoluzione contiene l'ordine delle righe dopo gli scambi;

`ipk`: contiene la permutazione al passo attuale, cioè ha scambiati gli elementi k (passo corrente) ed i ($i > k$, riga dell'elemento selezionato come pivot); viene reimpostato alla configurazione iniziale alla fine di ogni ciclo.

Il motivo dell'uso di due vettori è il seguente: la scrittura degli elementi nel vettore `z` viene ordinata per mezzo del vettore `ip`, ed anche gli elementi delle colonne di Z, che vengono letti ed inseriti nel vettore `c` (vettore, appunto, che contiene la colonna corrente), sono ordinati per `ip`. Così facendo, il primo punto del secondo passo avviene senza far uso di vettori di indici. Quando poi si effettua la ricerca del pivot all'interno del ciclo su k, non si può più usare il vettore `ip` (che comunque viene aggiornato per i passi successivi), ma si ha bisogno di un altro vettore che tenga traccia solo dell'ultimo scambio di righe, in modo da poter applicare l'ultima permutazione anche agli elementi già ordinati per `ip`.

Gli scambi attuati dal pivoting avvengono sulle righe della matrice Z, scambiando quindi l'ordine degli elementi all'interno di una stessa colonna, ma senza che essi vengano scambiati tra colonne differenti. Dal momento che ogni elemento di `b` necessita, per essere aggiornato, di un'intera colonna di Z, le componenti del vettore `b` non devono essere permutate ad ogni iterazione. Risulta invece necessario ordinare detto vettore secondo `ip` alla fine dell'algoritmo, in quanto l'aver scambiato le righe di Z, la quale contiene A^T, equivale ad aver scambiato le colonne di A e cioè l'ordine delle incognite nelle equazioni del sistema.

3.5 Utilizzo dei numeri complessi in C

Il linguaggio C, al contrario del Fortran, non dispone dell'aritmetica dei numeri complessi: si è quindi dovuto ovviare a questa mancanza scrivendo delle routine opportune.

All'inizio, sono state utilizzate le funzioni disponibili nei *Numerical Recipes* (cfr. [19, pg. 176-178, 948-950]), studiate appositamente per applicazioni numeriche in modo da ridurre possibili problemi dovuti all'aritmetica floating point.

Purtroppo, sebbene siano pensate per aumentare l'accuratezza dei calcoli, il fatto che debbano essere chiamate un numero considerevole di volte, porta ad un degrado inaccettabile delle prestazioni, a volte anche raddoppiando il tempo di esecuzione.

Il C, però, possiede un'interessante possibilità per ovviare a questo inconveniente: le *macro del preprocessore* (da ora chiamate solo *macro*); queste consentono di associare ad un nome una costante, un'istruzione o un insieme di istruzioni; nella fase precedente la compilazione, dove cioè interviene il preprocessore, tutte le occorrenze del nome di una macro vengono sostituite con il codice ad esso associato. L'utilizzo delle macro permette di mantenere il codice leggibile consentendo di svilupparlo modularmente, ma senza introdurre i rallentamenti dovuti alle chiamate a funzioni.

Poiché lo scopo non era quello di scrivere delle macro generali per le operazioni complesse ma solo quello di rendere efficienti i calcoli mantenendo il codice leggibile, si sono analizzate in dettaglio le operazioni che vengono svolte e si è notato come queste siano di tre tipi:

- $x \leftarrow -x$
- $x \leftarrow -x/y$
- $x \leftarrow x + y * z$

Si sono quindi modificate le routine citate in precedenza al fine di creare delle macro che svolgessero in un unico passo queste operazioni.

Oltre all'assenza dell'aritmetica complessa, il C è privo anche di una struttura dati primitiva per la gestione dei numeri complessi, per rappresentare i quali si è creato una struttura a due campi che contengano, rispettivamente, la parte reale e la parte immaginaria del numero.

Ancora nei *Numerical Recipes* viene indicata una strategia per la gestione in maniera efficiente di vettori di numeri complessi (cfr. [19, pg. 24, 507-508]). Questa si basa sull'idea di gestire esplicitamente ogni numero complesso come parte reale e parte immaginaria (senza utilizzare strutture ausiliarie come `struct`), dichiarando un vettore di n complessi come un vettore di $2n$ numeri floating point, per poi idealmente utilizzare l'array come se fosse diviso in coppie che rappresentano, ciascuna, un numero complesso.

Sebbene venisse prospettato un incremento nelle prestazioni, esso non c'è stato, anzi, in alcuni casi, il metodo rivisto secondo queste indicazioni è risultato più lento di quello originario di circa il 10%. Si è mantenuta, pertanto, la versione senza queste modifiche.

3.6 Conversione implicita di `float` a `double` in C

Nella versione tradizionale del linguaggio C di Kernighan e Ritchie [16], le variabili `float` sono automaticamente convertite al tipo `double` prima di *ogni* operazione, comprese quindi quelle aritmetiche ed il passaggio dei parametri a funzione. In questo modo tutta l'aritmetica è eseguita in doppia precisione:

infatti, tutte le funzioni delle librerie standard C per le operazioni aritmetiche sono di tipo `double` e vengono quindi eseguite in doppia precisione. Nel caso in cui il risultato dell'operazione debba essere assegnato ad una variabile di tipo `float`, la precisione "in eccesso" viene scartata.

Questo comportamento non è più vero in generale, in quanto l'ANSI C [17] stabilisce che l'aritmetica sugli operatori `float` *può* avvenire in singola precisione, invece che in doppia; dal momento che è solo una possibilità, spesso i compilatori fanno uso esclusivo dell'aritmetica in doppia precisione, se non specificato altrimenti.

Il modo di operare della prima edizione del C consente di semplificare l'architettura delle librerie standard, in quanto è necessario fornire una sola implementazione di ogni funzione, e solitamente un po' di precisione in più non provoca alcun danno. Purtroppo, questa conversione automatica non consente di scrivere codici numerici efficienti, in quanto esiste una precisa ragione algoritmica per avere singola o doppia precisione, anche per una questione di massimizzazione delle performance: si vedano per esempio i grafici 3.7 e 3.8 del tempo di esecuzione delle routine per il metodo di Wassing con e senza le ottimizzazioni (`-O3`) attivate, le quali disabilitano la conversione automatica.

Si vede bene in figura 3.7 che i tempi delle routine utilizzanti la singola precisione sono molto simili a quelle che usano la doppia precisione. Al contrario, in figura 3.8 i tempi risultano coerenti con le aspettative, ovvero, i tempi di esecuzione in singola precisione sono approssimativamente dimezzati, rispetto alle analoghe in doppia precisione.

Ove possibile, si consiglia di eliminare la conversione automatica in quanto

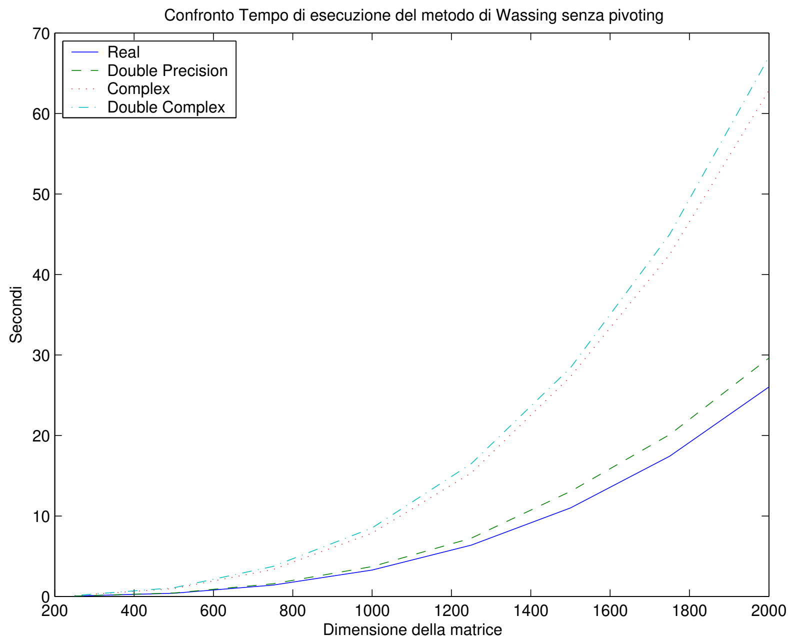

Figura 3.7: Tempo di esecuzione senza l'uso di `-O3`

un codice scritto per la singola precisione risulta essere lento quasi quanto uno scritto in doppia precisione, ma con una accuratezza nettamente minore.

3.7 Utilizzo delle ottimizzazioni

Si usano ancora le figure 3.7 e 3.8 per mostrare una importante caratteristica del compilatore `gcc/g77`: si noti, infatti, il miglioramento notevole delle prestazioni (anche del 50%) attivando l'ottimizzazione `-O3` del compilatore (esistono diversi livelli di ottimizzazione, per approfondimenti si consulti `man gcc`).

Le ottimizzazioni `-Ox`, con `x=1, 2, 3`, raggruppano diverse opzioni di compilazione rivolte principalmente a migliorare le prestazioni nella gestione

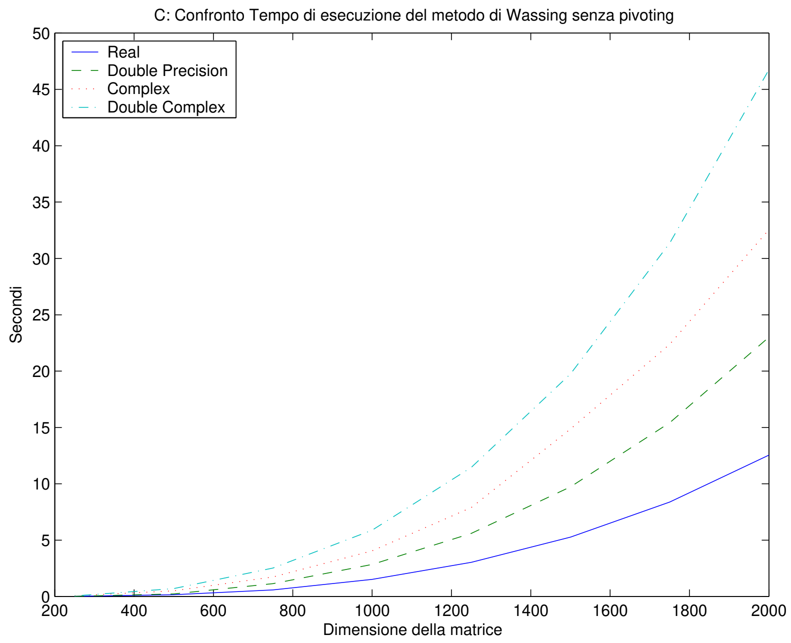

Figura 3.8: Tempo di esecuzione con l'uso di `-O3`

della memoria, nell'esecuzione di operazioni matematiche e nell'ordine di esecuzione delle istruzioni all'interno del programma. La meno aggressiva delle tre è `-O1`, mentre quella che consente i maggiori guadagni, a fronte di un tempo di compilazione maggiore, è `-O3` (l'opzione utilizzata per tutti i test svolti).

L'utilizzo delle ottimizzazione è vivamente consigliato, in quanto con lo stesso codice sorgente vengono generati eseguibili più performanti. Tuttavia, questo è raccomandabile soltanto ad un punto dello sviluppo ormai maturo: infatti, il guadagno ottenibile potrebbe mascherare parti di codice poco efficienti, che invece potrebbero essere identificate senza l'utilizzo delle ottimizzazioni.

3.8 Complessità

Esaminiamo ora la complessità del metodo di Wassing e quindi quali prestazioni ci si possono attendere; nel far questo verranno prese in considerazione solo le operazioni floating point, in quanto molto più lente di quelle tra interi.

Riguardando alla descrizione del metodo della sezione 2.2, si può notare come la parte che contribuisce maggiormente alla complessità dell'algoritmo è principalmente la *2)*, in quanto l'ultimo passo non introduce praticamente nessuna operazione di rilievo, mentre nel punto *1)* vengono eseguite n operazioni[2] dovute alle divisioni necessarie alla modifica della prima colonna, del tutto trascurabili rispetto alle operazioni della *2)*

Analizziamo dunque in dettaglio questa sezione, esaminando i vari passi che lo compongono:

a) $\forall i = k, k+1, \ldots, n, n+1$

$$z_{ik} = z_{ik} + \sum_{j=1}^{k-1} z_{ij} z_{jk}$$

Si eseguono $n-k+2$ somme per modificare gli elementi z_{ik} aggiungendovi la sommatoria, la quale richiede $k-2$ somme e $k-1$ moltiplicazioni, per un totale di $(n-k+2)(2k-3+1) = (n-k+2)(2k-2)$ operazioni.

b) $\forall i = k+1, k+2, \ldots, n, n+1$

$$z_{ik} = -z_{ik}/z_{kk}$$

Per modificare un elemento si ha bisogno di eseguire una sola operazione, la divisione. Si ha dunque un totale di $(n-k+1)$ operazioni per l'aggiornamento degli altrettanti elementi.

[2]Le operazioni unarie, come il cambio si segno, in genere non vengono contate.

c) $\forall j = 1, 2, \ldots, k-1$ e $\forall i = k+1, k+2, \ldots, n, n+1$

$z_{ij} = z_{ij} + z_{ik} z_{kj}$

Si eseguono due operazioni per aggiornare ogni elemento che si trova nella sotto-matrice di indici $(i, j) = (k+1 \ldots n+1, 1 \ldots k-1)$; sono pertanto necessarie $2(k-1)(n-k+1)$ operazioni per completare l'aggiornamento.

I tre punti analizzati in precedenza, si trovano all'interno di un ciclo su k che varia tra 2 ed n; si dovranno dunque sommare questi contributi al variare di k (alcuni termini verranno omessi in quanto non contribuiscono a determinare la complessità dell'algoritmo ed i passaggi fra un'equazione e la successiva non avranno il simbolo di uguaglianza ma quello di approssimazione):

$$\sum_{k=2}^{n} \Big[(n-k+2)(2k-2) + (n-k+1) + 2(k-1)(n-k+1)\Big] =$$

$$= \sum_{k=2}^{n} \Big[2k - 2 + (n-k+1)(2k-2+1+2k-2)\Big] \approx$$

$$\approx \sum_{k=2}^{n} \Big[(n-k+1)(4k-3)\Big] \approx \sum_{k=2}^{n} \Big[4kn - 4k^2\Big] = 4n \sum_{k=2}^{n} k - 4 \sum_{k=2}^{n} k^2 \approx$$

$$\approx \frac{4n^2(n+1)}{2} - \frac{4n(n+1)(2n+1)}{6} \approx 2n^3 - \frac{4}{3}n^3 = \frac{2}{3}n^3$$

Nel caso l'algoritmo utilizzato faccia uso del pivoting parziale, si devono anche considerare i confronti, che hanno costo proporzionale a $\mathcal{O}(n^2)$, mentre in caso non si faccia uso del pivoting i confronti sono proporzionali a $\mathcal{O}(n)$.

La complessità risulta essere, pertanto, identica a quella del metodo di eliminazione di Gauss, con eventuale pivoting. Alle stesse conclusioni si perviene nel caso di più termini noti, ovvero, se si hanno m termini noti, allora si ottiene una complessità di

$$\frac{2}{3}n^3 + 2mn^2$$

flops.

Capitolo 4

Sperimentazione e risultati

Il metodo di Wassing è stato implementato:

1. con e senza pivoting;
2. in singola e doppia precisione;
3. in aritmetica reale e complessa;
4. in linguaggio Fortran e C.

Le routine sviluppate sono state confrontate con le omologhe routine LAPACK [1] [2], che è la libreria standard di comune utilizzo per le applicazioni scientifiche ed industriali, nelle quali sia richiesta la risoluzione di sistemi lineari a matrice densa.

4.1 Piattaforma hardware

Il computer su cui sono stati eseguiti tutti i test ha le seguenti caratteristiche:

Processore: Pentium4 1.8GHz, 512KB cache;

Memoria: 512MB;

S.O.: Gentoo GNU/Linux, kernel 2.4.20-gentoo-r6;

gcc/g77: versione 3.2.3 20030422;

glibc: versione 2.3.2-r3;

Librerie LAPACK: versione 3.0;

presso il Centro di Calcolo del Dipartimento di Matematica "Ulisse Dini".

4.2 Sistemi lineari di test

Verranno ora descritti i sistemi lineari utilizzati nei test delle performance e sull'accuratezza.

Come evidenziato in maggior dettaglio nella sezione 5.2, verrà presa in considerazione la matrice A^T e non direttamente A, matrice dei coefficienti del sistema lineare.

La matrice utilizzata nel caso di numeri reali è la matrice di Hilbert con la diagonale principale incrementata di 1:

$$\begin{aligned} i = j \quad &\rightarrow \quad a_{ij} = 1 + \frac{1}{i+j-1} \\ i \neq j \quad &\rightarrow \quad a_{ij} = \frac{1}{i+j-1} \end{aligned}$$

Questa matrice è a diagonale dominante (oltre che simmetrica e definita positiva) e risulta ben condizionata, come si vede in figura 4.1.

La matrice viene utilizzata nel caso di numeri reali per i test di performance ed in quelli di accuratezza che non implementano il pivoting.

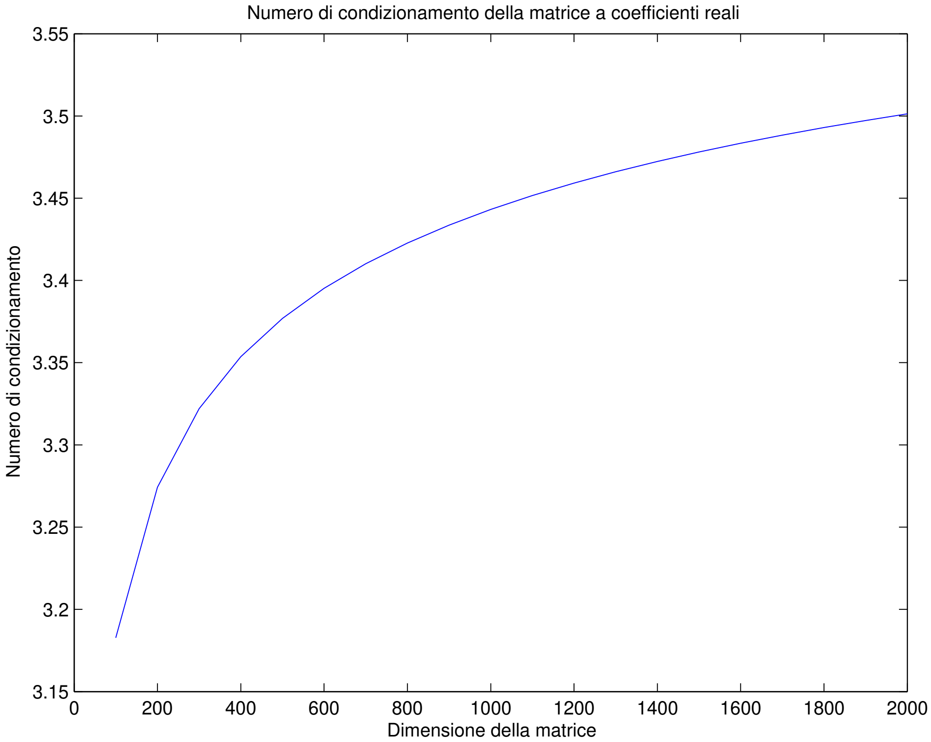

Figura 4.1: Numero di condizionamento per la matrice a coefficienti reali

Nel caso di numeri complessi[1], la precedente matrice viene modificata in modo opportuno:

$$i = j \quad \rightarrow \quad a_{ij} = \left(1 + \frac{1}{i+j-1},\ 0\right)$$

$$i < j \quad \rightarrow \quad a_{ij} = \left(0,\ \frac{1}{i+j-1}\right)$$

$$i > j \quad \rightarrow \quad a_{ij} = \left(0,\ -\frac{1}{i+j-1}\right)$$

[1]Per rappresentare i numeri complessi si utilizzerà la forma $(a, b) \equiv z = a + ib$, una coppia di numeri il cui primo elemento è la parte reale e il secondo la parte immaginaria.

Il numero di condizionamento di questa matrice, mostrato in figura 4.2, è basso, indice di come essa sia ben condizionata.

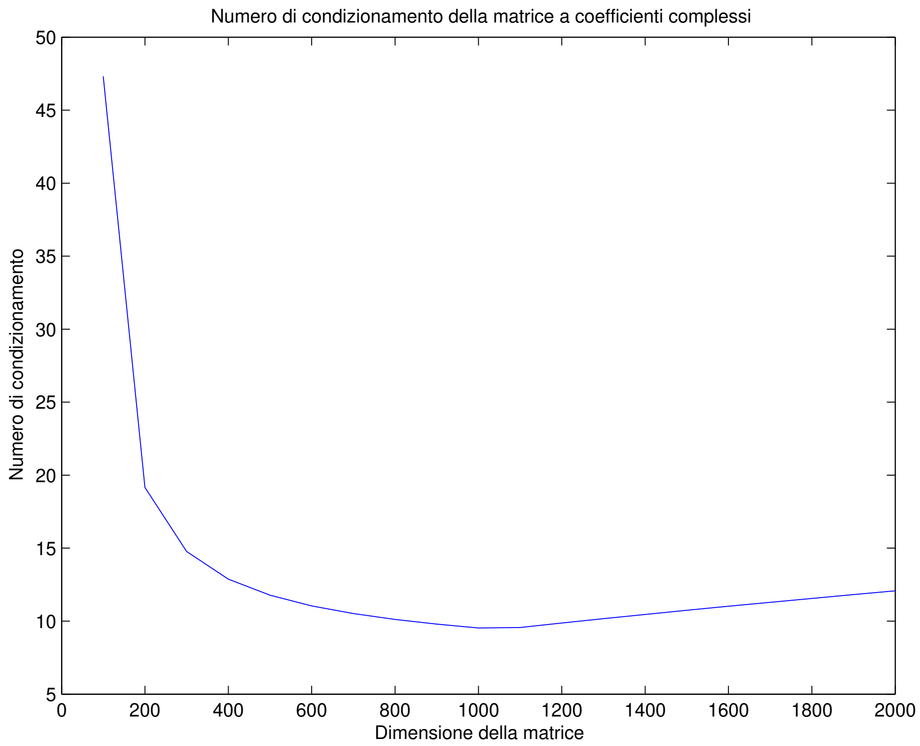

Figura 4.2: Numero di condizionamento per la matrice a coefficienti complessi

Per verificare l'accuratezza dei metodi con pivoting parziale, sono state utilizzate le seguenti matrici:

- numeri reali:

$$\begin{aligned} i+j=n+1 \quad &\rightarrow \quad a_{ij} = 1 + \frac{1}{n-i+j} \\ else \quad &\rightarrow \quad a_{ij} = \frac{1}{n-i+j} \end{aligned}$$

- numeri complessi:

$$
\begin{array}{lcl}
i+j=n+1 & \rightarrow & a_{ij} = \left(1+\dfrac{1}{n-i+j},\ 0\right) \\
i+j \leq n & \rightarrow & a_{ij} = \left(0,\ -\dfrac{1}{n-i+j}\right) \\
else & \rightarrow & a_{ij} = \left(0,\ \dfrac{1}{n-i+j}\right)
\end{array}
$$

Queste matrici sono ottenute dalle precedenti invertendo l'ordine delle righe: la prima diventa l'ultima, la seconda la penultima, e così via; le matrici sono antidiagonali dominanti ed è quindi necessario applicare il pivoting per ottenere dei risultati corretti.

Il termine noto, per i test di performance, non è particolarmente importate, ed il vettore b è stato scelto come composto da tutti 3 per i numeri reali, e da $(3, 0)$ per i numeri complessi.

Per i test di accuratezza, invece, la scelta del termine noto diventa più importante: ogni elemento del vettore b è stato scelto come la somma degli elementi sulla riga omologa della matrice dei coefficienti, in modo tale, pertanto, da avere, come soluzione, il vettore $x = (1, \ldots, 1)^T$.

4.3 Modalità dei confronti

I test eseguiti sono suddivisi in due categorie:

1. test delle performance;

2. test sull'accuratezza.

I codici sono stati tutti compilati attivando le ottimizzazioni di compilazione `-O3` e la generazione, l'esecuzione e l'analisi dei risultati dei test sono state automatizzate tramite l'utilizzo di alcuni script.

Per i test delle performance, l'algoritmo che implementa il metodo di Wassing con pivoting viene confrontato con la routine LAPACK di risoluzione di sistemi lineari generali, `gesv`. Gli algoritmi senza pivoting, invece, sono stati confrontati, oltre che con `gesv`, anche con la routine per sistemi lineari con la matrice dei coefficienti simmetrica e definita positiva `posv` che, tuttavia, ha una complessità di $\frac{1}{3}n^3$ flops, invece che $\frac{2}{3}n^3$. I test con le funzioni `posv` di LAPACK sono state svolte solo per numeri reali, in quanto la matrice complessa utilizzata non è definita positiva per $n \geq 68$.

I test sull'accuratezza confrontano i metodi con e senza pivoting con la routine LAPACK `gesv`; viene poi confrontata l'accuratezza delle implementazioni in Fortran e C.

4.4 Test delle performance

Dal momento che l'obiettivo principale dell'algoritmo è il risparmio di memoria, è prevedibile una riduzione della velocità di esecuzione rispetto a metodi classici come l'eliminazione di Gauss.

Per stimare questa perdita si è confrontato la velocità di esecuzione del metodo di Wassing in relazione ad analoghe routine LAPACK: si è fatta variare la dimensione del sistema da risolvere tra $n = 250$ ed $n = 2000$ con incrementi di 250. Ogni esecuzione è stata ripetuta 30 volte e dei tempi viene presa la media, per cercare di ridurre l'influenza di eventuali altri programmi e/o servizi in quel momento in esecuzione.

Dal momento che l'algoritmo implementato consente la risoluzione simultanea di più sistemi lineari aventi la stessa matrice dei coefficienti, si sono volute stimare le performance anche in questo caso.

Il numero di termini noti, nb, è stato fatto variare come $nb = 5, 10, 15, \ldots,$ 50, mentre la dimensione del sistema è mantenuta costante, e pari a $n = 2000$, in modo da avere differenze più marcate nei tempi con differenti valori di nb. Anche in questo caso, le esecuzioni sono state eseguite 30 volte, prendendo poi la media dei tempi.

Per monitorare il tempo di esecuzione si è utilizzato l'utility `time` (anche se è stato necessario ricompilarla, in quanto quella disponibile sul sistema di test dava alcuni problemi). La scelta di utilizzare un programma esterno, rispetto alle istruzioni messe a disposizione dal Fortran e dal C, è stata fatta per evitare che metodi differenti di acquisizione del tempo da parte dei due linguaggi avessero ripercussioni sull'analisi a-posteriori dei risultati.

4.4.1 Analisi dei risultati

I grafici vengono presentati in tabelle di quattro immagini, in modo da poter raggruppare le presentazioni in maniera organica: le figure sono infatti riunite al variare del tipo di dato, del linguaggio di programmazione e delle routine impiegate. Lo schema di presentazione è il seguente:

tab. 4.1-4.2: tempi di esecuzioni della routine per il metodo di Wassing e LAPACK, in Fortran e C: ogni grafico mostra i tempi di esecuzione al variare del tipo di dato;

tab. 4.3-4.4: confronto dei tempi di esecuzione delle routine per il metodo di Wassing nelle implementazioni Fortran e C: ogni grafico mostra i tempi di esecuzione delle due implementazioni, divisi per tipo di dato, con e senza pivoting;

tab. 4.5-4.6: confronti tra il metodo di Wassing, con e senza pivoting, con le routine LAPACK corrispondenti, in Fortran e C;

tab. 4.7: rapporto dei tempi di esecuzioni del metodo di Wassing e delle routine LAPACK nelle implementazioni in Fortran e C;

tab. 4.8-4.9: confronto dei tempi di esecuzioni delle routine LAPACK nelle implementazioni in Fortran e C;

tab. 4.10..4.12: confronto dei tempi di esecuzione del metodo di Wassing e delle routine LAPACK nel caso di più termini noti.

Come ci si poteva aspettare, il tempo di esecuzione delle routine LAPACK non cambia nelle due implementazioni (tabelle 4.8 e 4.9).

Più interessante è osservare il comportamento delle routine per il metodo di Wassing in Fortran e C: mentre per la doppia precisione i tempi sono praticamente uguali (sebbene il C risulti leggermente più lento), lo stesso non è più vero per la singola precisione. Questo comportamento è evidente nelle tabelle 4.3 e 4.4, ed ancora più chiaramente in tabella 4.7. Si può notare, a titolo di esempio, come l'implementazione C per i numeri complessi in singola precisione risulti essere meno performante (attorno al 20%) rispetto a quella Fortran: questo può essere dovuto all'assenza dell'aritmetica complessa in C, cui si è sopperito tramite funzioni aggiuntive. Questa differenza viene meno nel caso di doppia precisione, forse anche per la maggior lentezza di esecuzione di queste operazioni.

Nel caso di più termini noti, al crescere del loro numero, il tempo di esecuzione aumenta di conseguenza anche se, come era prevedibile, la parte che maggiormente contribuisce alle performance rimane quella di fattorizzazione della matrice. Inoltre, si può notare in tabella 4.12 come, al crescere del numero di termini noti nel sistema, le prestazioni del metodo di Wassing e delle routine LAPACK tendano ad avvicinarsi, anche se lentamente.

Infine, nelle tabelle 4.5 e 4.6 si ha il confronto di prestazioni tra il metodo implementato e le routine LAPACK. Come ci si attendeva, le prestazioni del codice per il metodo di Wassing risultano minori di quelle LAPACK, a causa dei maggiori spostamenti dei dati in memoria. Tuttavia, la perdita di prestazioni, che è al massimo del 50%, viene controbilanciata da un utilizzo 4 volte inferiore di memoria.

4.5 Verifica dell'accuratezza

L'accuratezza di un codice di calcolo è uno dei suoi aspetti fondamentali. Talora è preferibile utilizzare uno strumento che necessiti di un maggior tempo di esecuzione ma che garantisca una maggiore accuratezza dei risultati. È dunque necessario verificare l'accuratezza dei codici su problemi test opportuni.

L'accuratezza che possiamo ottenere nella risoluzione di un sistema lineare

$$Ax = b$$

dipende dal numero di condizionamento, $\kappa(A)$, della matrice A. A parità di numero di condizionamento (ovvero di problema da risolvere) è quindi interessante confrontare l'accuratezza ottenuta con i vari codici.

Per far questo si hanno a disposizione due possibilità: studiare l'errore ($err = x - \hat{x}$) oppure studiare il residuo ($r = b - A\hat{x}$); la seconda opzione è stata scartata in quanto presenta due problemi, uno di carattere teorico (è possibile avere un residuo "piccolo" con un errore molto grande, cfr. [5, pg. 181] [24] [26]) ed uno di carattere pratico (è necessario eseguire esplicitamente il prodotto tra la matrice dei coefficienti e la soluzione calcolata e questo potrebbe introdurre ulteriori errori). Si è dunque optato per l'analisi dell'errore err.

Ottenuta dunque la soluzione $\hat{x}$ tramite il metodo numerico, è possibile valutare il corrispondente vettore di errore $err = x - \hat{x}$. Calcolando, allora, $\|err\|_2 = \|x - \hat{x}\|_2$ si ottiene una misura dell'errore commesso durante la risoluzione.

Sebbene ci si aspetti che le componenti del vettore err siano piccole, al crescere della dimensione del vettore, anche $\|err\|_2$ tenderà a crescere di conseguenza (è infatti una somma di quantità non negative), rendendo difficile un confronto al variare della dimensione del sistema; per ovviare a questo inconveniente si è presa in considerazione un'altra quantità:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n} err_i^2} = \frac{\|err\|_2}{\sqrt{n}}$$

che normalizza la misura dell'errore, in relazione alla dimensione del problema (cfr. [5, pg. 86]), ed a cui si farà riferimento in seguito col nome di *norma 2 "pesata"*.

Oltre a questo indicatore, si è anche voluto stimare l'errore massimo commesso durante la risoluzione del sistema lineare, cioè il più grande scostamento di una componente di $\hat{x}$ dall'omologa componente del vettore x:

$$\max_{i=1,\dots,n} |x_i - \hat{x}_i| = \|err\|_\infty.$$

4.5.1 Analisi dei risultati

Le modalità di rappresentazione dei risultati sono simili a quanto già visto nella sezione 4.4.1: i grafici sono raggruppati in gruppi di quattro, al variare del tipo di dato. Viene mostrata la norma 2 "pesata" e la norma infinito del metodo di Wassing in tutte le varianti implementate, confrontandole sia tra di loro, che con le analoghe routine LAPACK. La presentazione avviene in quest'ordine:

tab. 4.13-4.14: Fortran, confronto di norma 2 e norma infinito tra il metodo di Wassing senza pivoting e la routine LAPACK `gesv`;

tab. 4.15-4.16: Fortran, confronto di norma 2 e norma infinito tra il metodo di Wassing con pivoting e la routine LAPACK `gesv`;

tab. 4.17-4.18: C, confronto di norma 2 e norma infinito tra il metodo di Wassing senza pivoting e la routine LAPACK `gesv`;

tab. 4.19-4.20: C, confronto di norma 2 e norma infinito tra il metodo di Wassing con pivoting e la routine LAPACK `gesv`;

tab. 4.21..4.24: confronto tra le implementazioni Fortran e C del metodo di Wassing senza pivoting e routine LAPACK `gesv`;

tab. 4.25..4.28: confronto tra le implementazioni Fortran e C del metodo di Wassing con pivoting e routine LAPACK `gesv`;

Un fatto molto importante da notare è la quasi assoluta coincidenza dei risultati, riguardo all'accuratezza, del metodo di Wassing nei due linguaggi utilizzati (tant'è che si era pensato di proporre soltanto i risultati di un'implementazione e quelli di confronto): nel caso dei numeri reali, i grafici sono perfettamente sovrapponibili in tutti i casi analizzati, mentre per i numeri complessi le differenze sono più evidenti nel caso di utilizzo dell'algoritmo con pivoting (tabelle 4.25 e 4.27) rispetto a quello senza (tabelle 4.21 e 4.23).

La precisione delle routine LAPACK, come ci si poteva attendere, rimane la stessa nel linguaggio Fortran e C; le librerie, infatti, sono sì scritte in Fortran, ma sono già compilate, quindi nei due linguaggi si fa uso della stessa identica routine.

Di particolare interesse è il confronto tra la precisione del metodo di Wassing e delle routine LAPACK nel risolvere il medesimo problema. In generale, l'accuratezza risulta essere simile, eccezion fatta nel caso di numeri complessi negli algoritmi che implementano il pivoting: la routine `gesv` risulta, infatti, essere più precisa di circa una cifra decimale.

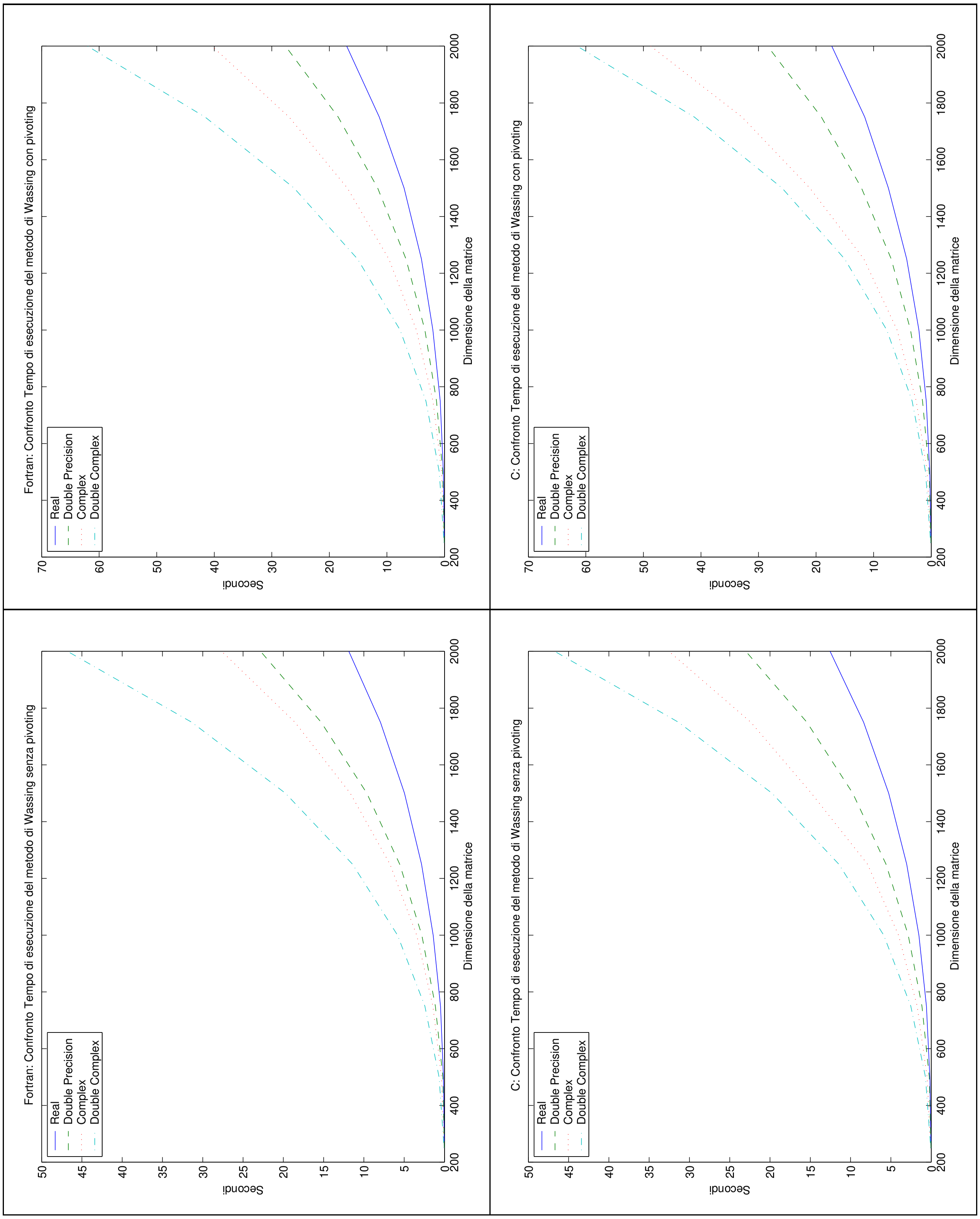

Tabella 4.1: Tempi di esecuzione delle routine wa[pvt]sv, Fortran e C

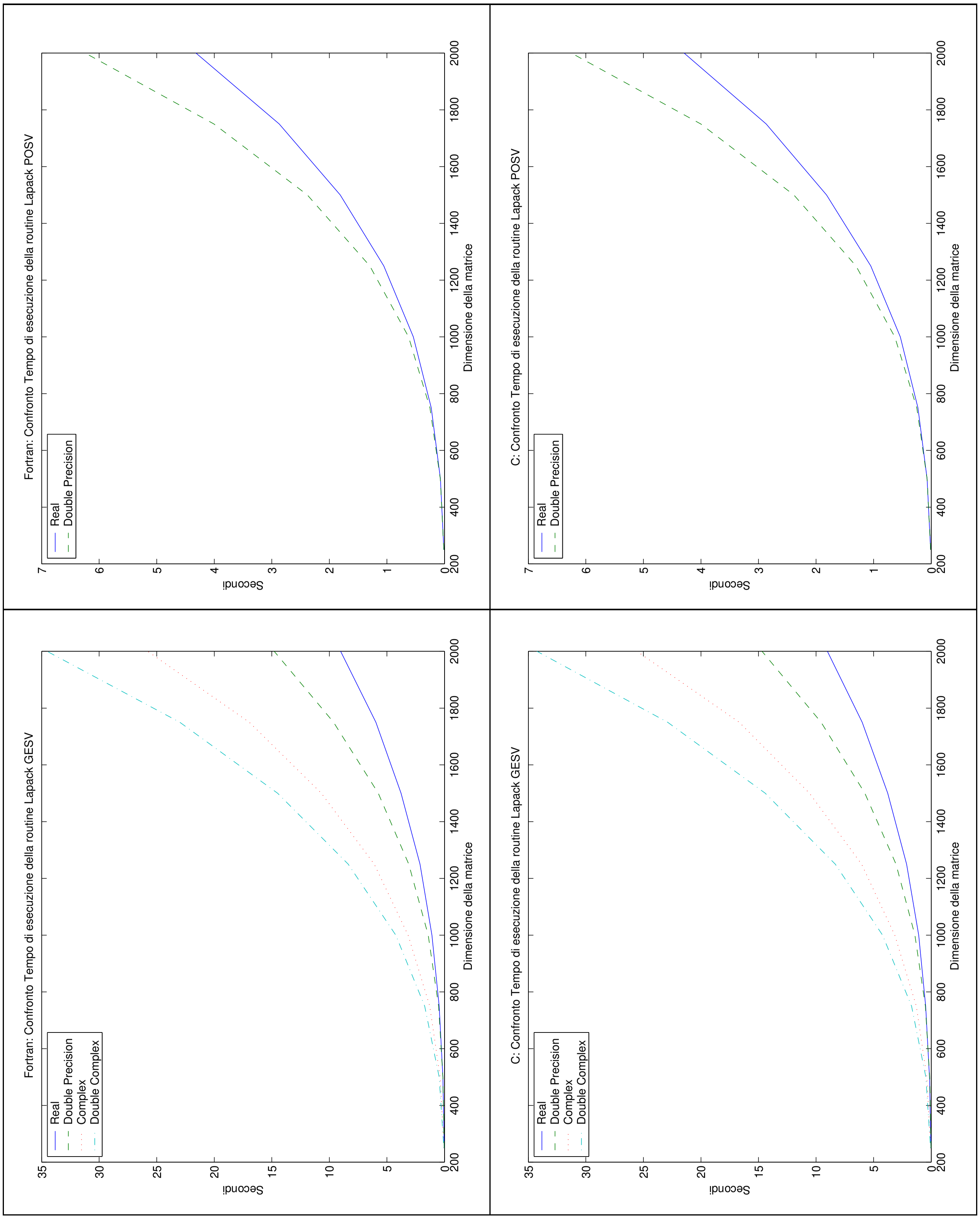

Tabella 4.2: Tempi di esecuzione delle routine LAPACK `gesv` e `posv`

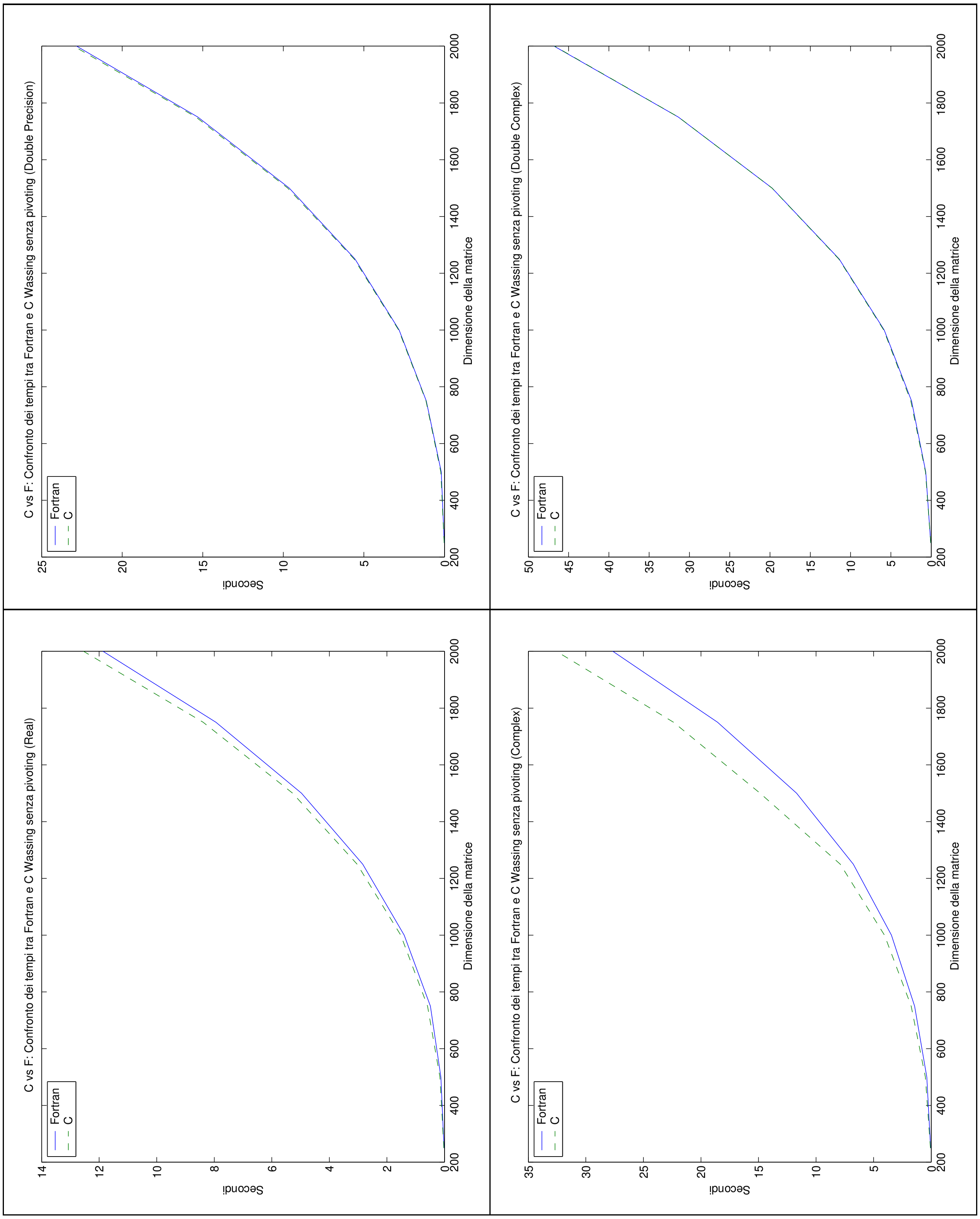

Tabella 4.3: Confronto `wasv` Fortran e C

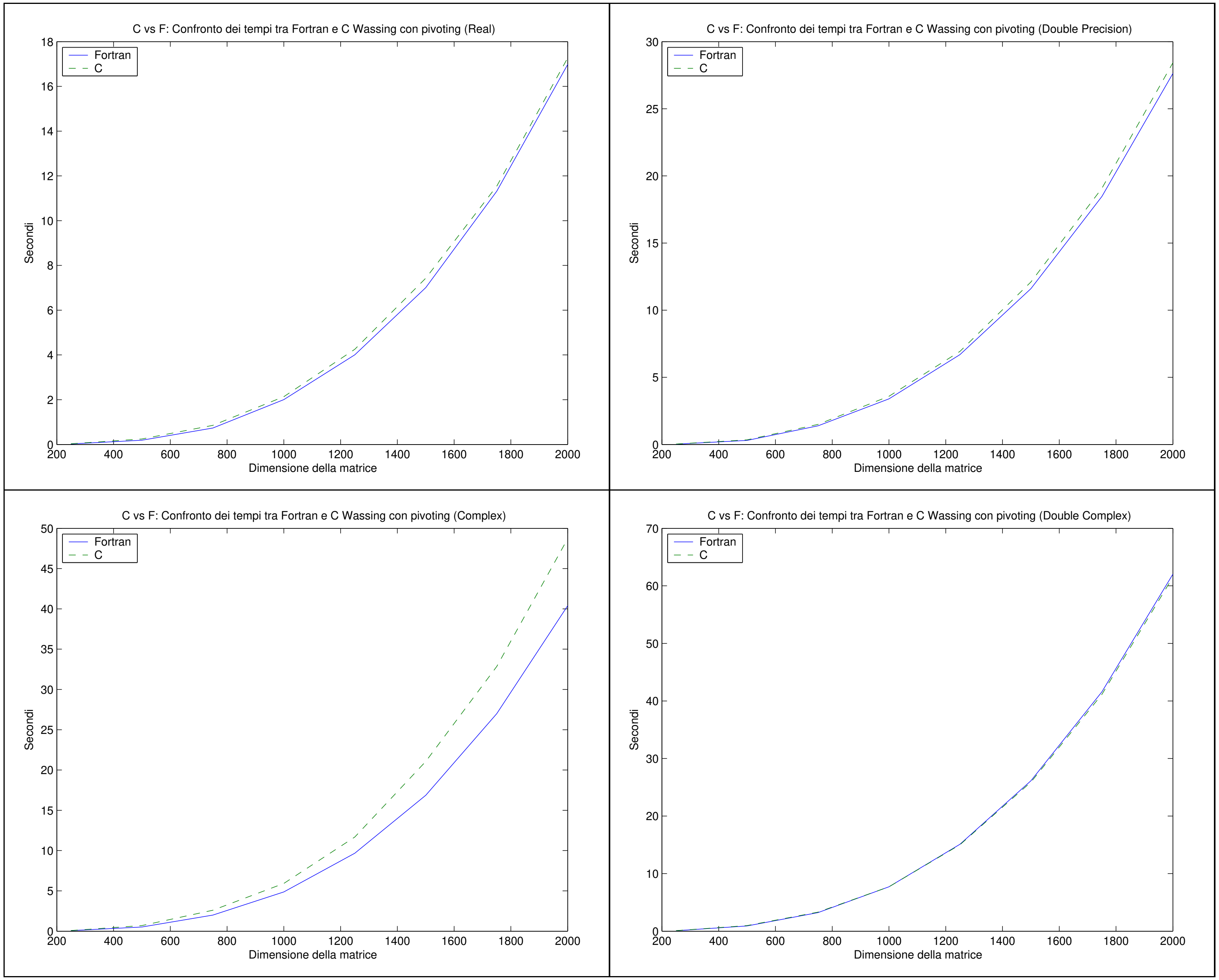

Tabella 4.4: Confronto `wapvtsv` Fortran e C

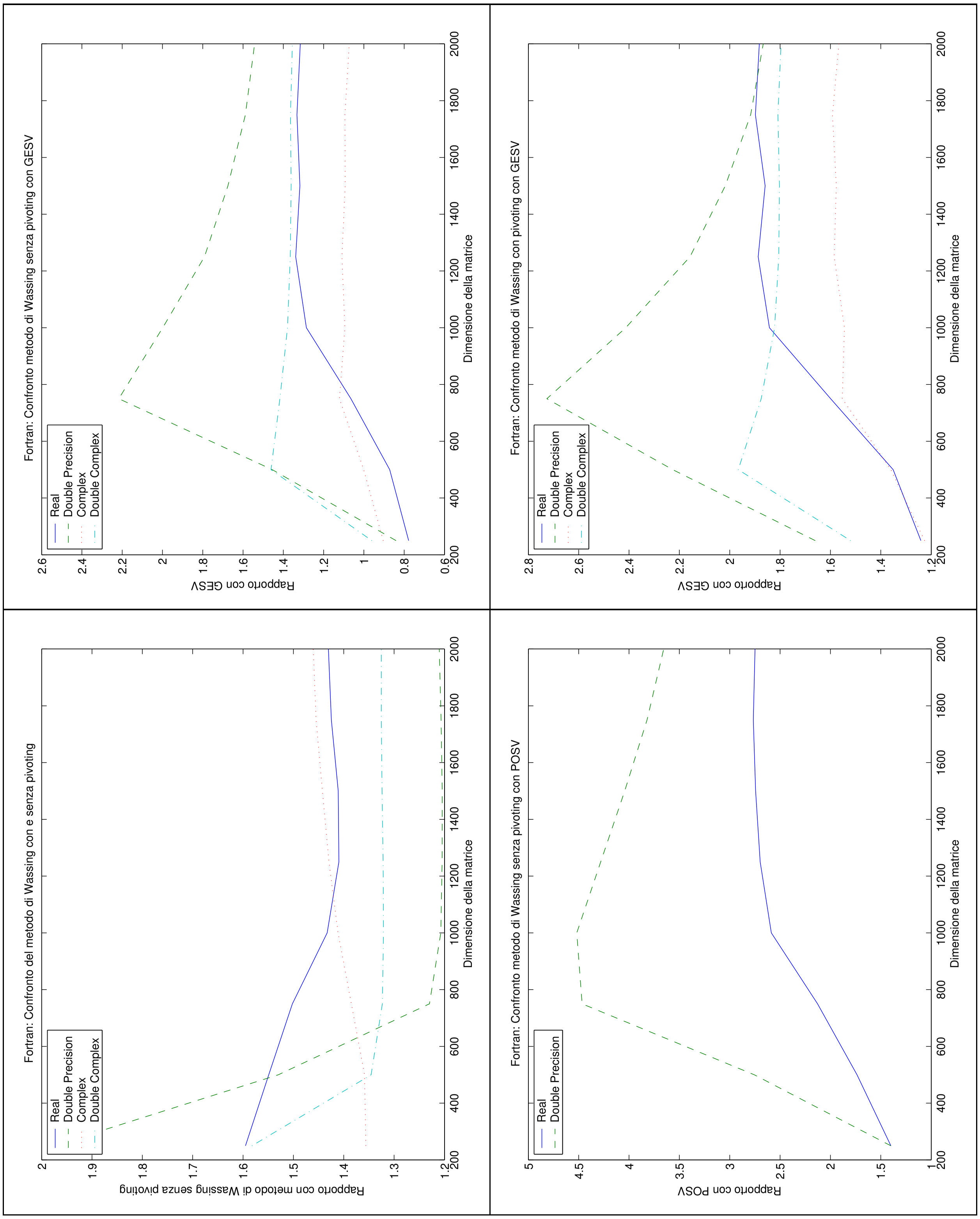

Tabella 4.5: Fortran, Confronti tra `wasv`, `wapsvsv`, `gesv` e `posv`

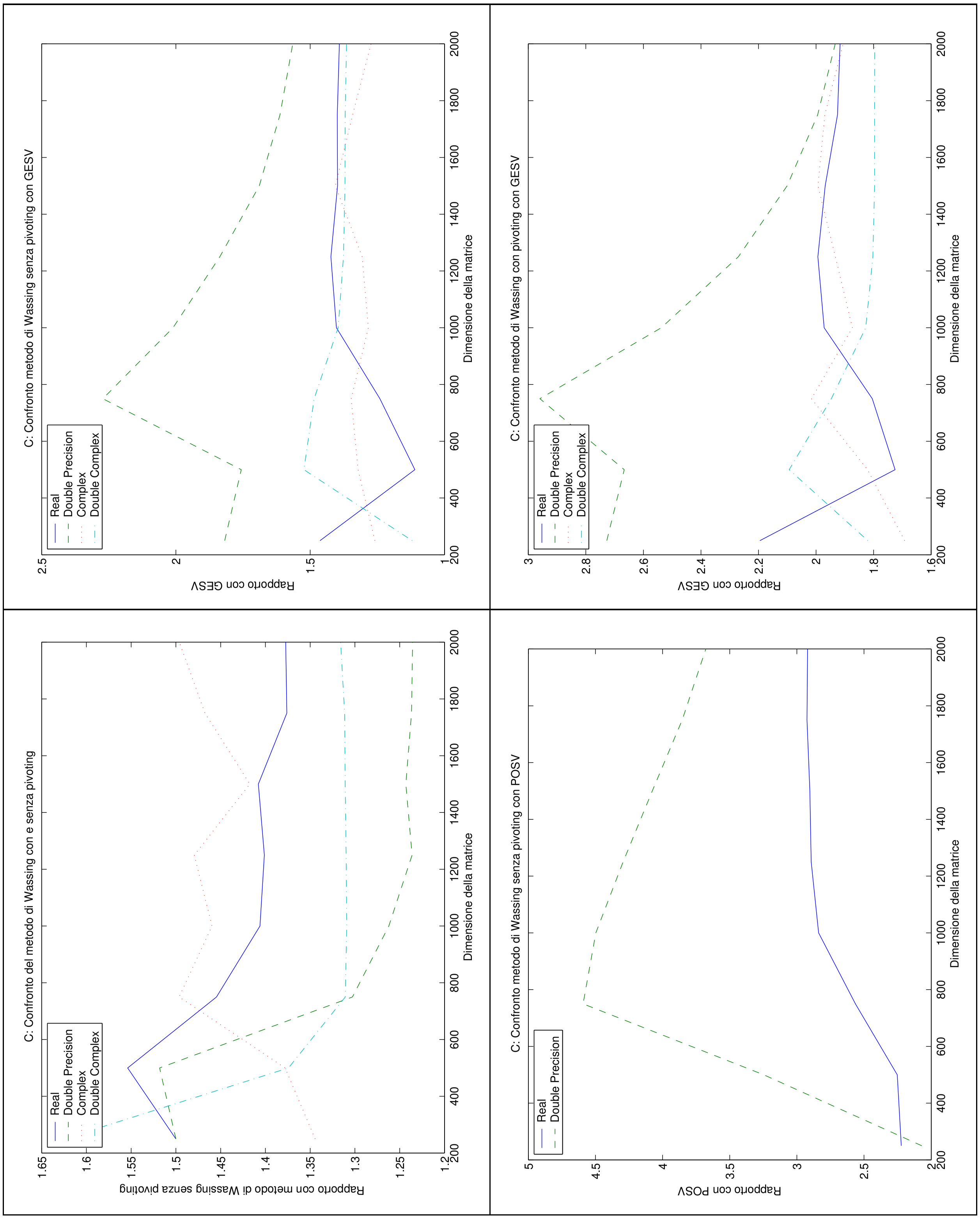

Tabella 4.6: C, Confronti tra `wasv`, `wapsvsv`, `gesv` e `posv`

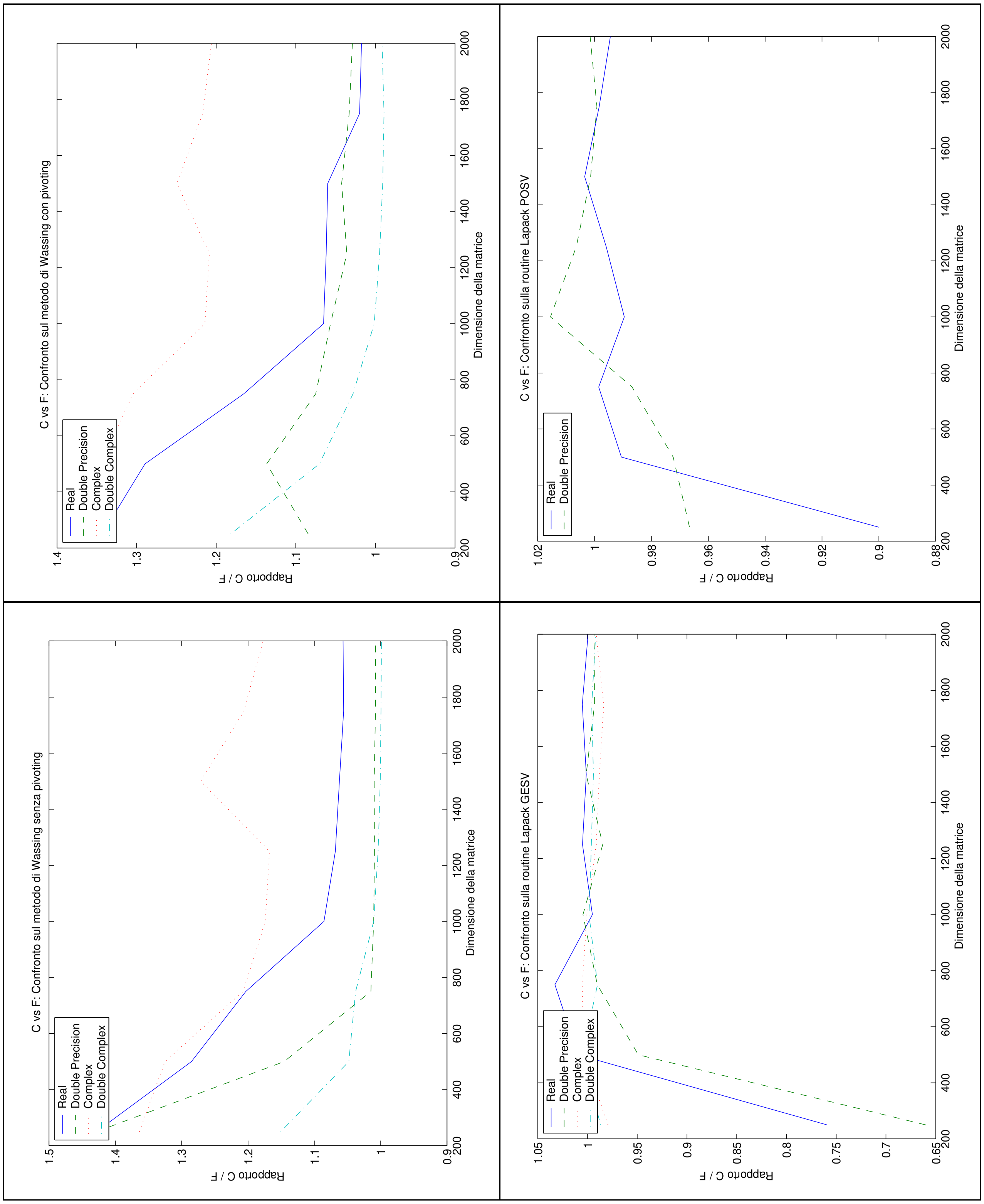

Tabella 4.7: Rapporto dei tempi di esecuzione in Fortran e C delle routine utilizzate

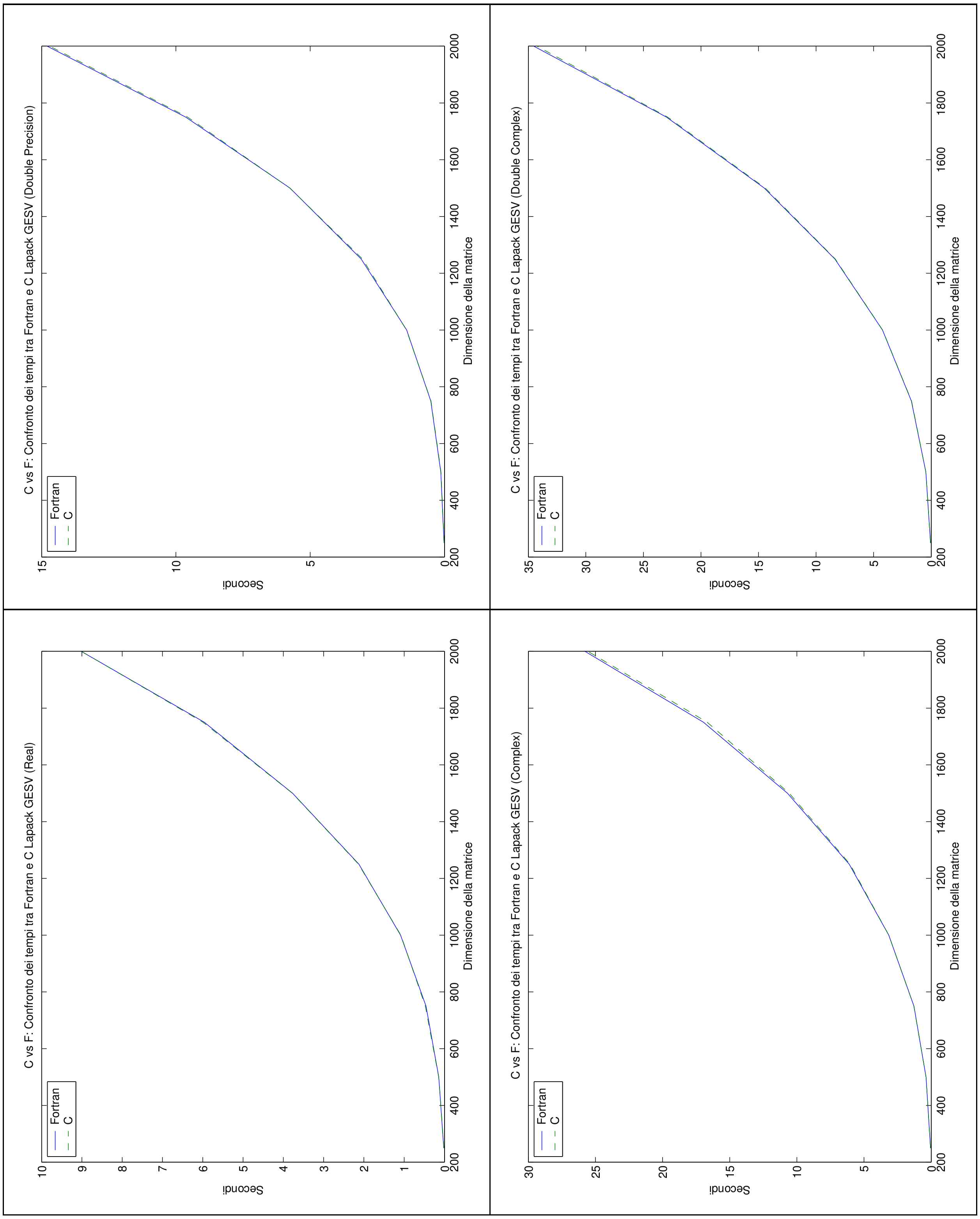

Tabella 4.8: Confronti `gesv` in Fortran e C

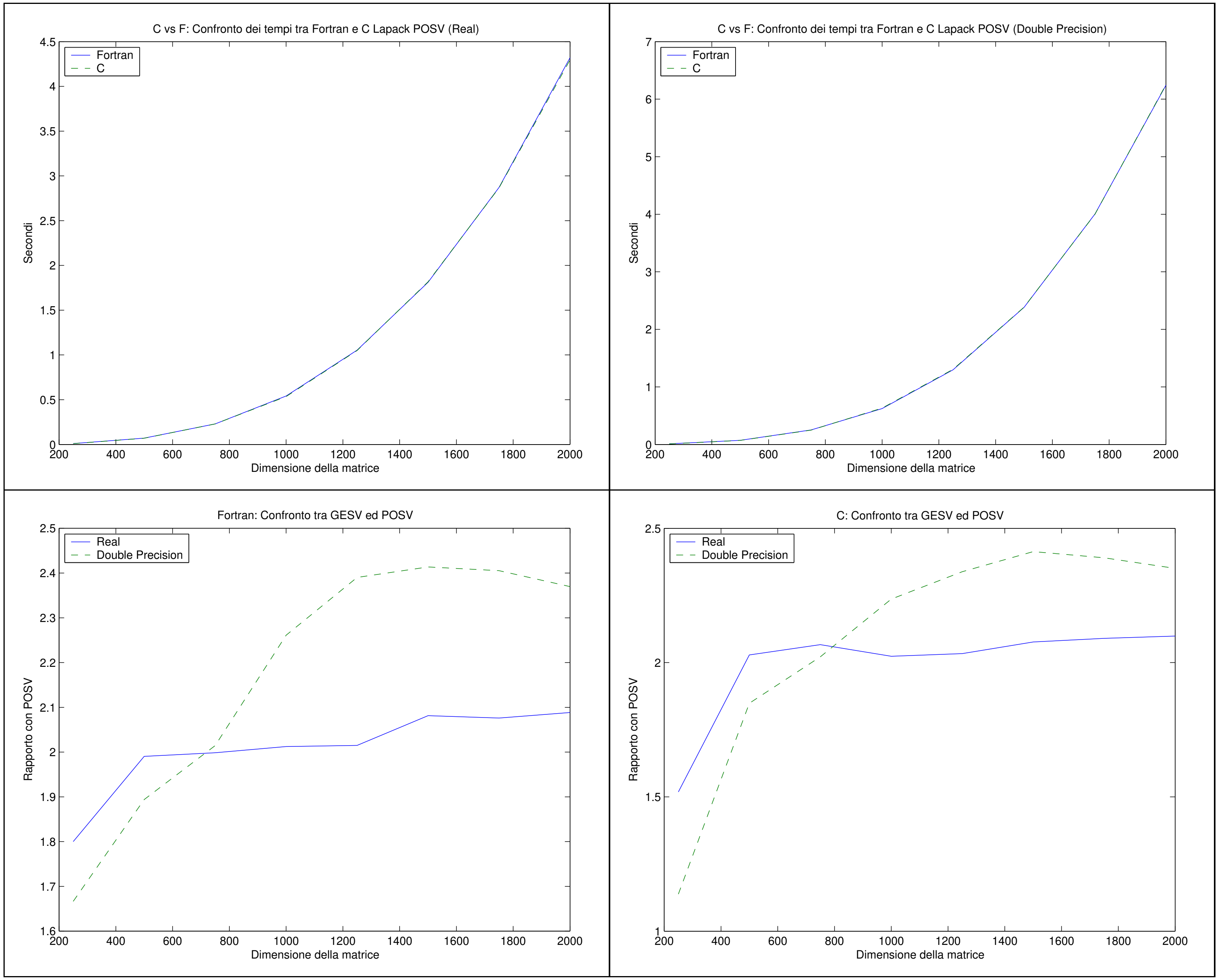

Tabella 4.9: Confronti posv in Fortran e C e rapporto tra gesv e posv

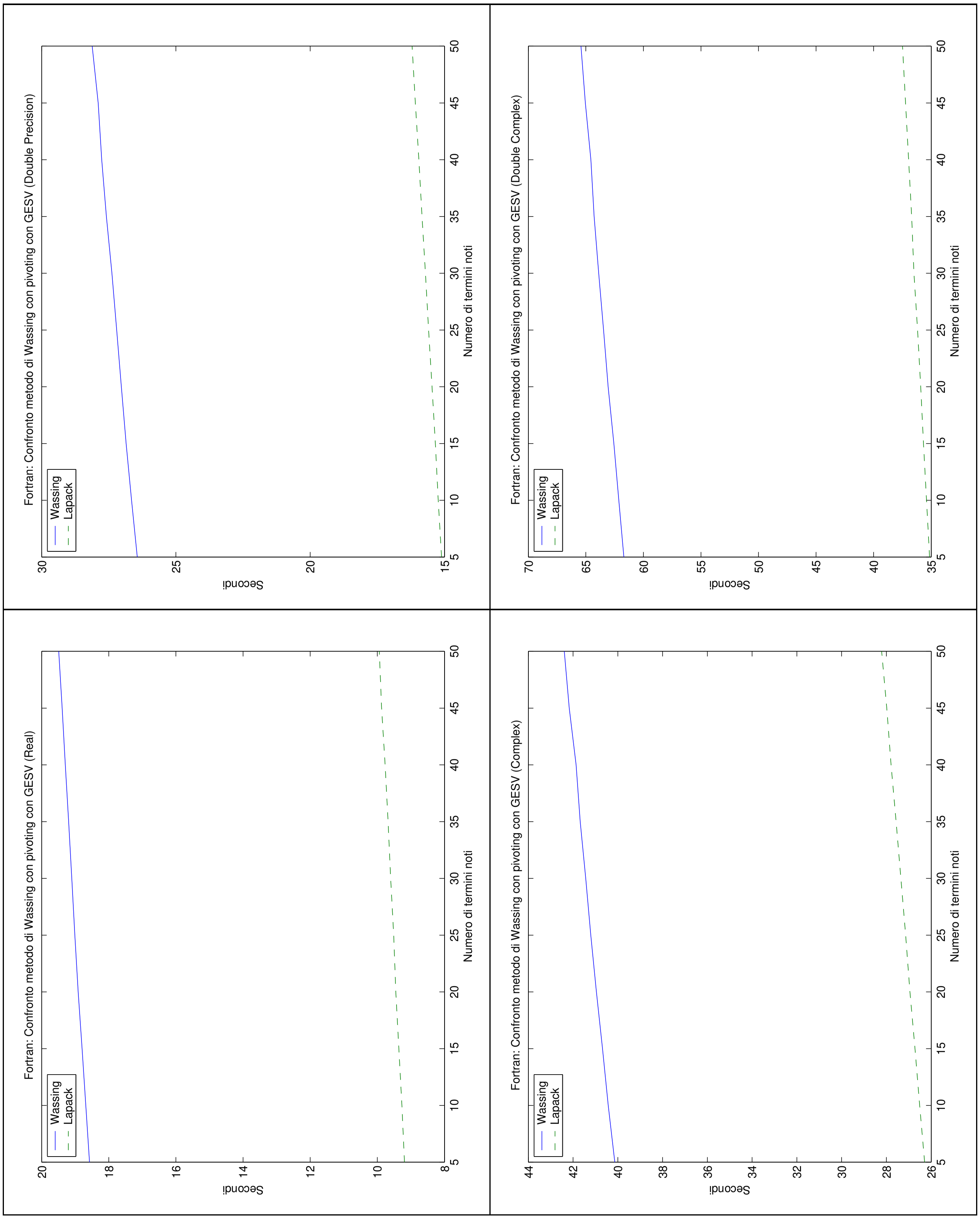

Tabella 4.10: Più termini noti, Fortran, Confronto tra `wapvtsv` e `gesv`

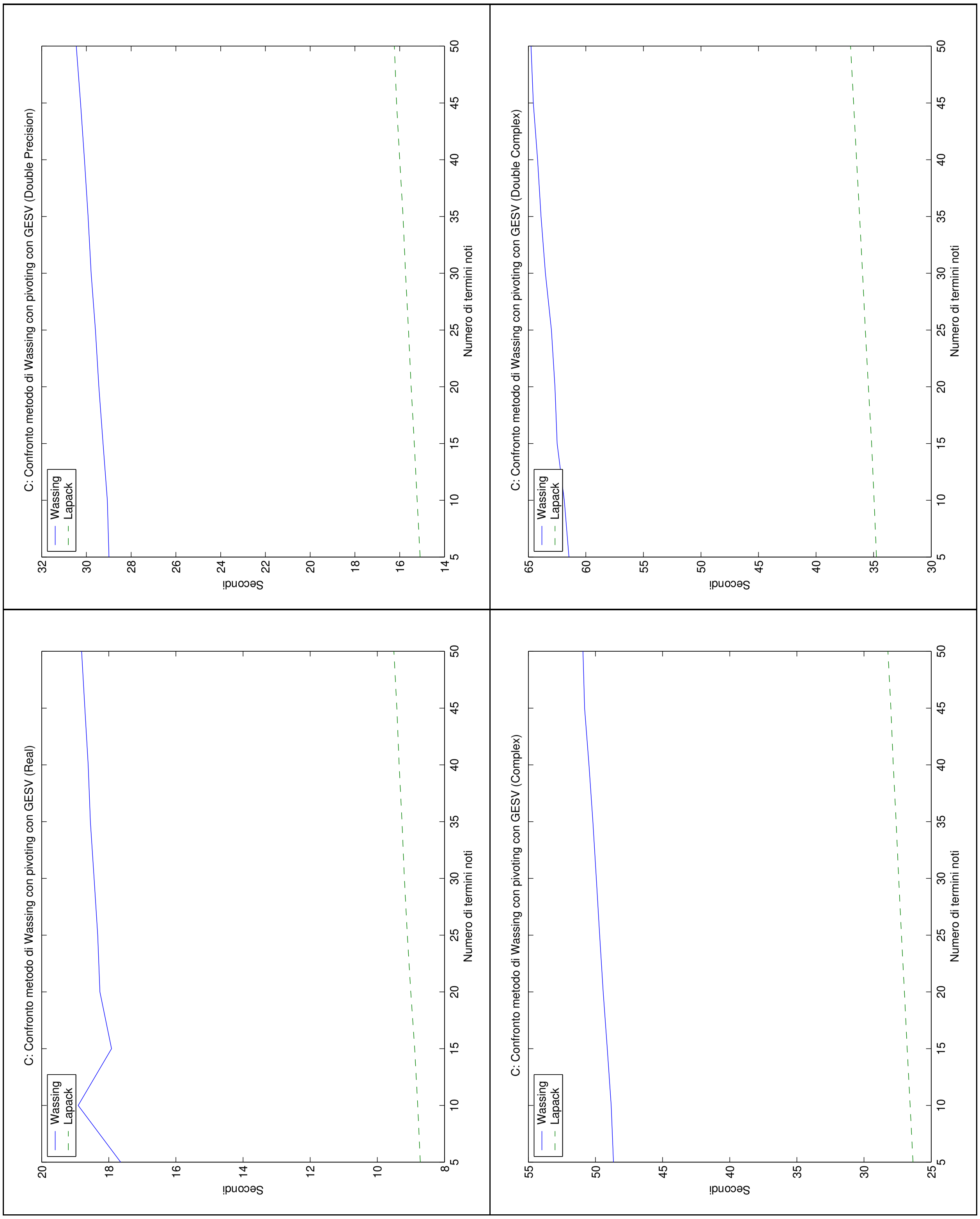

Tabella 4.11: Più termini noti, C, Confronto tra `wapvtsv` e `gesv`

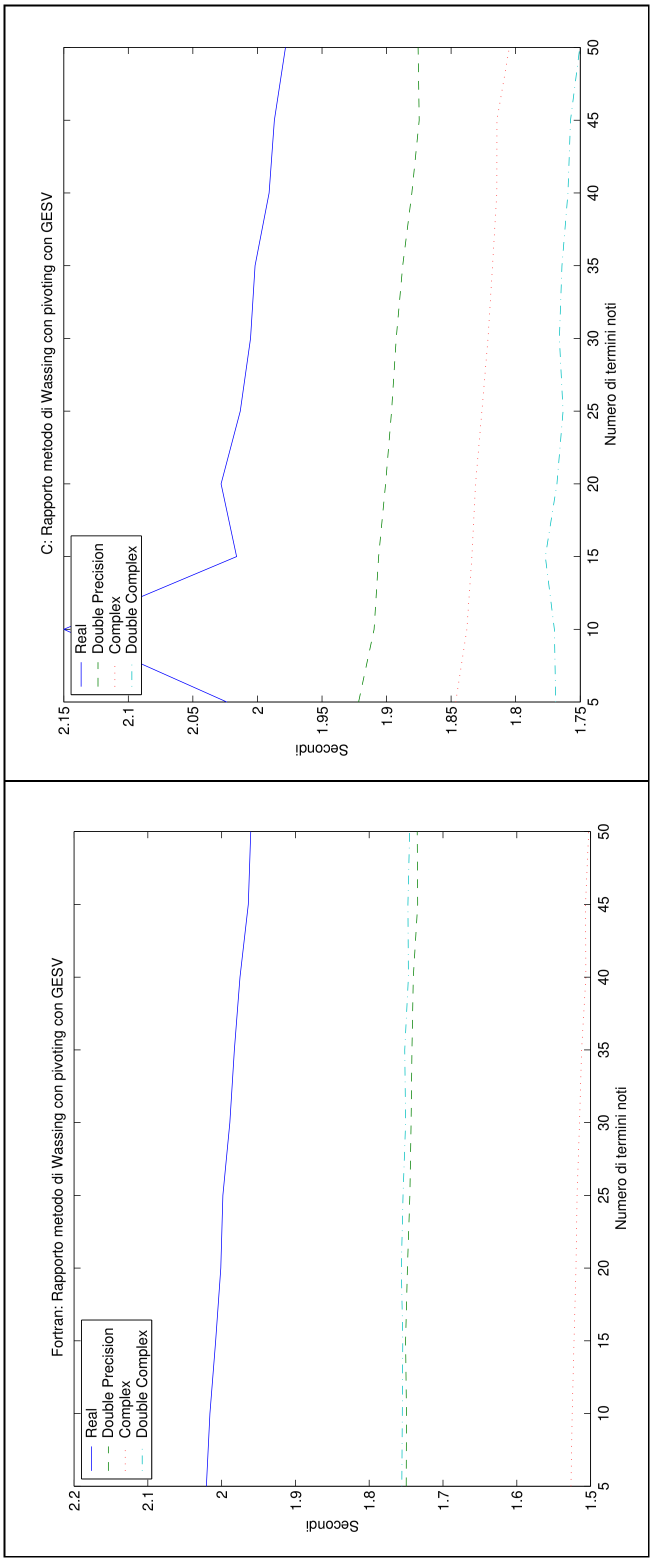

Tabella 4.12: Più termini noti, Rapporto tra `wapvtsv` e `gesv`

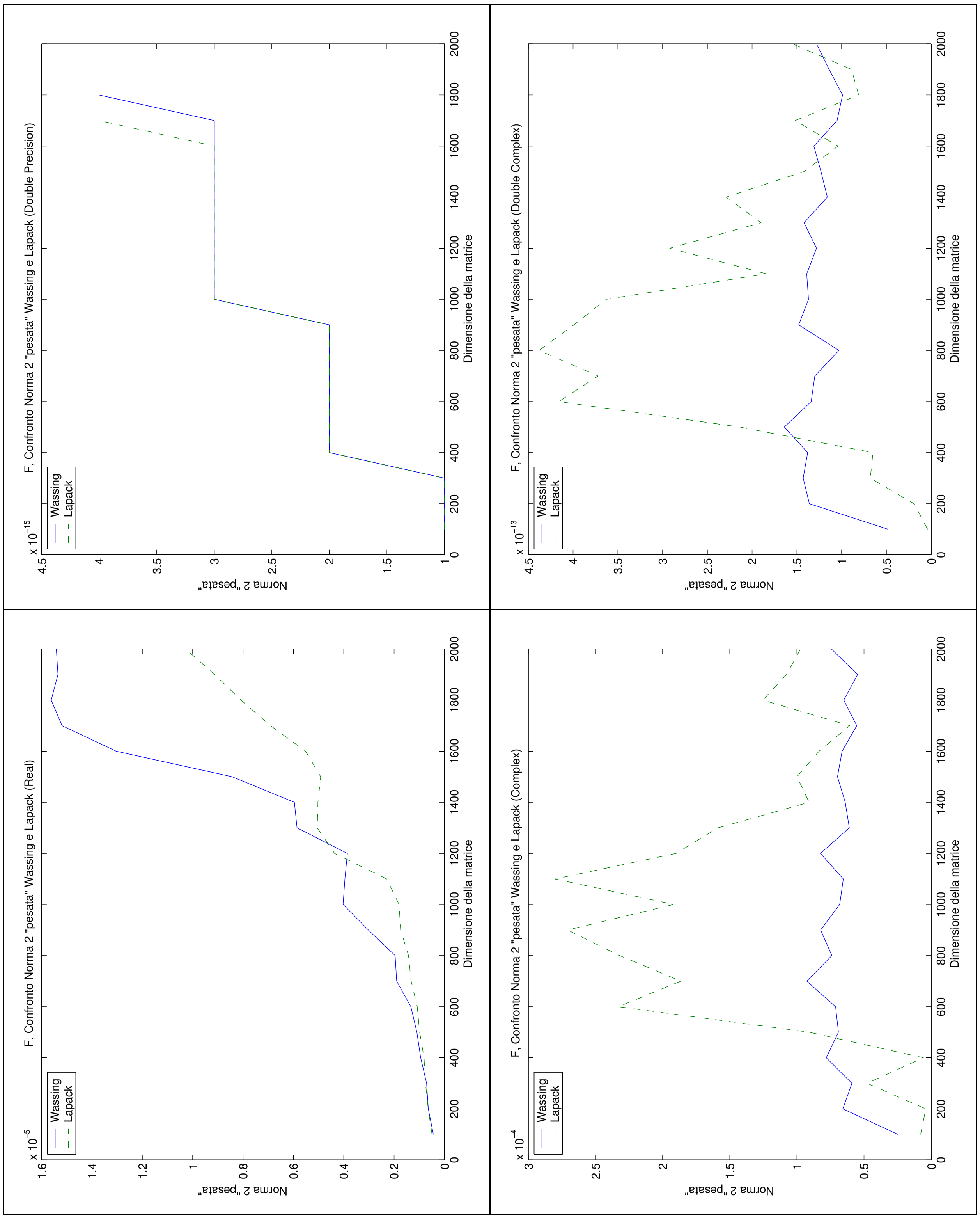

Tabella 4.13: Precisione, Fortran, Confronto norma 2 “pesata” `wasv` e `gesv`

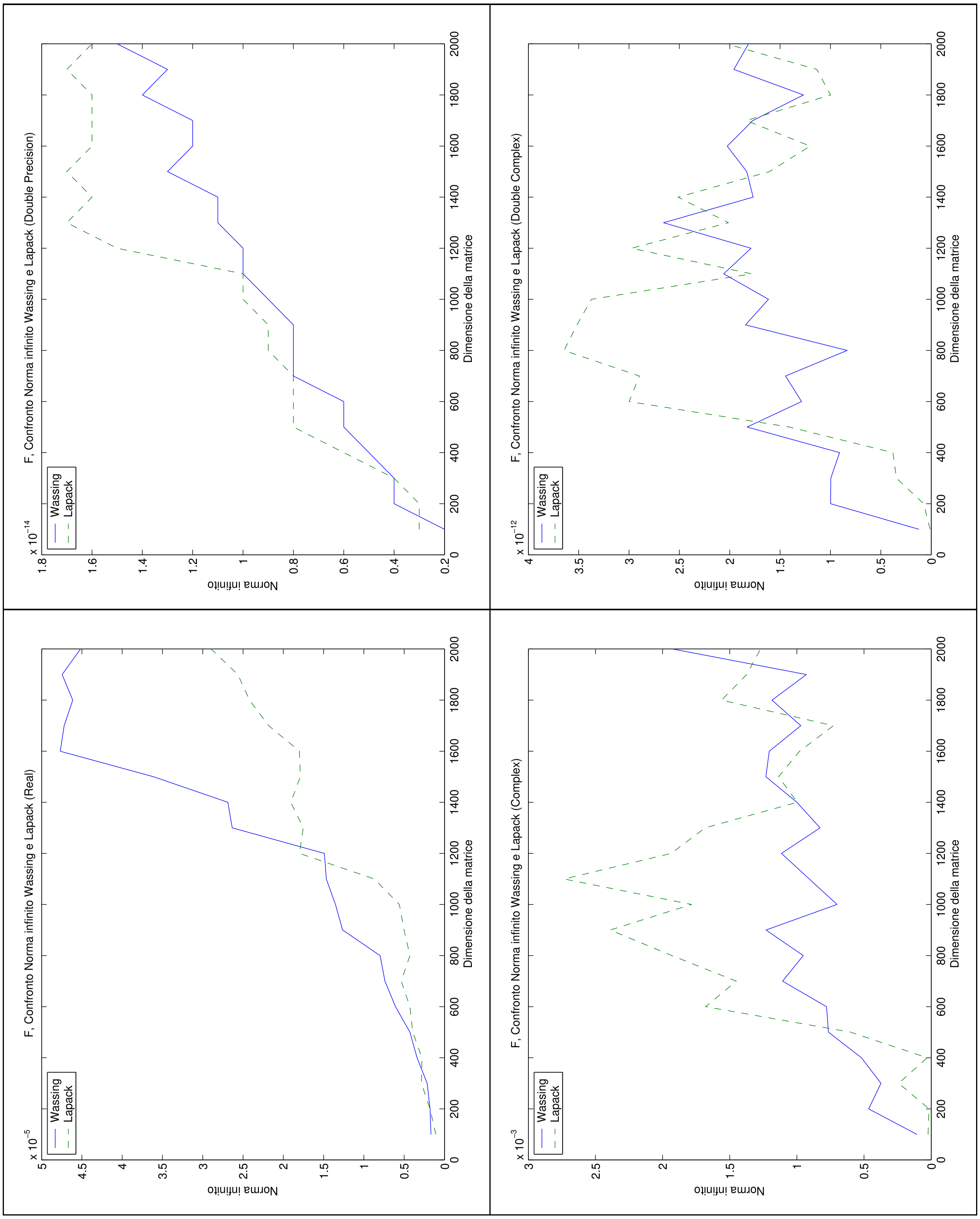

Tabella 4.14: Precisione, Fortran, Confronto norma infinito `wasv` e `gesv`

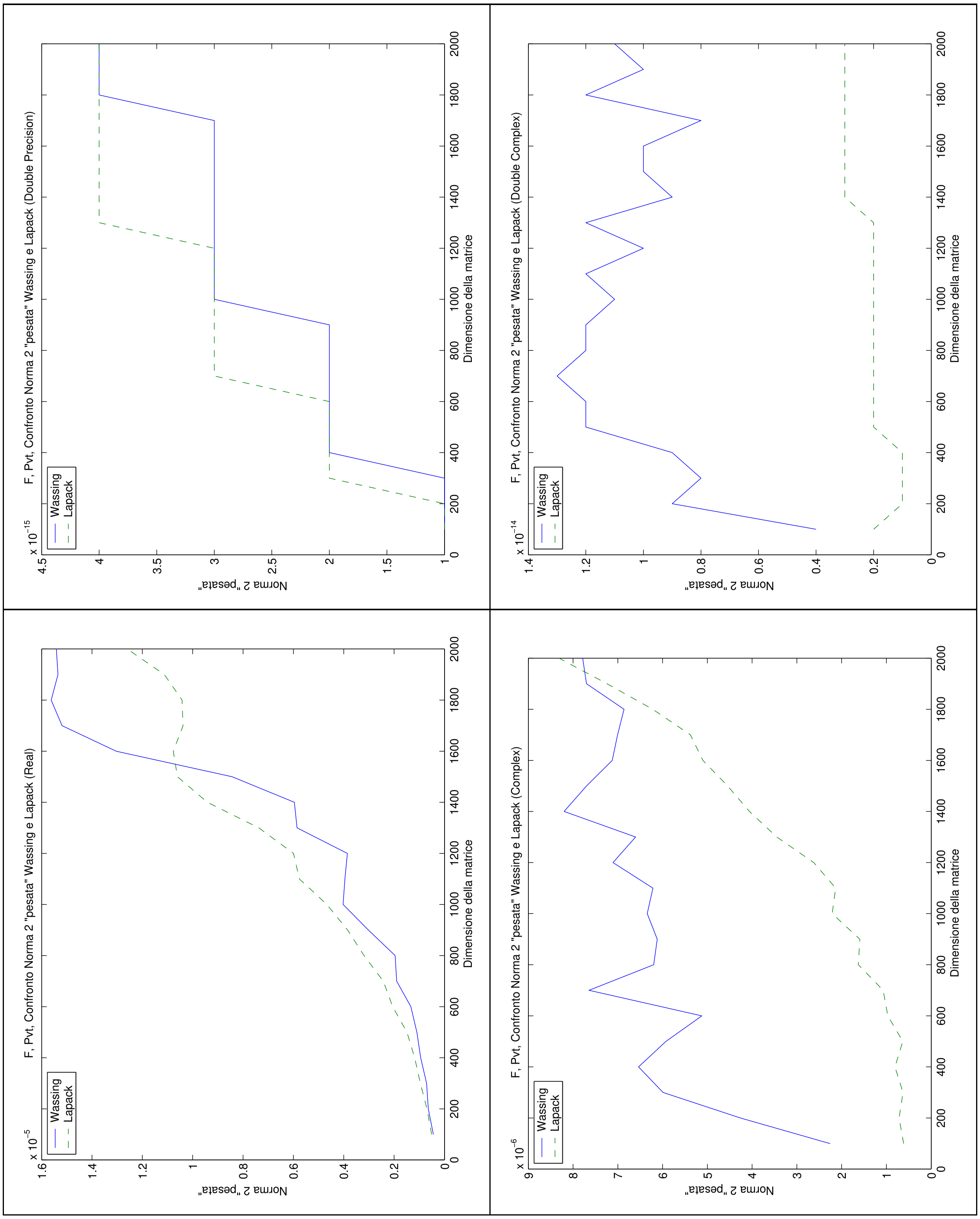

Tabella 4.15: Precisione, Fortran, Confronto norma 2 “pesata” `wapvtsv` e `gesv`

Tabella 4.16: Precisione, Fortran, Confronto norma infinito `wapvtsv` e `gesv`

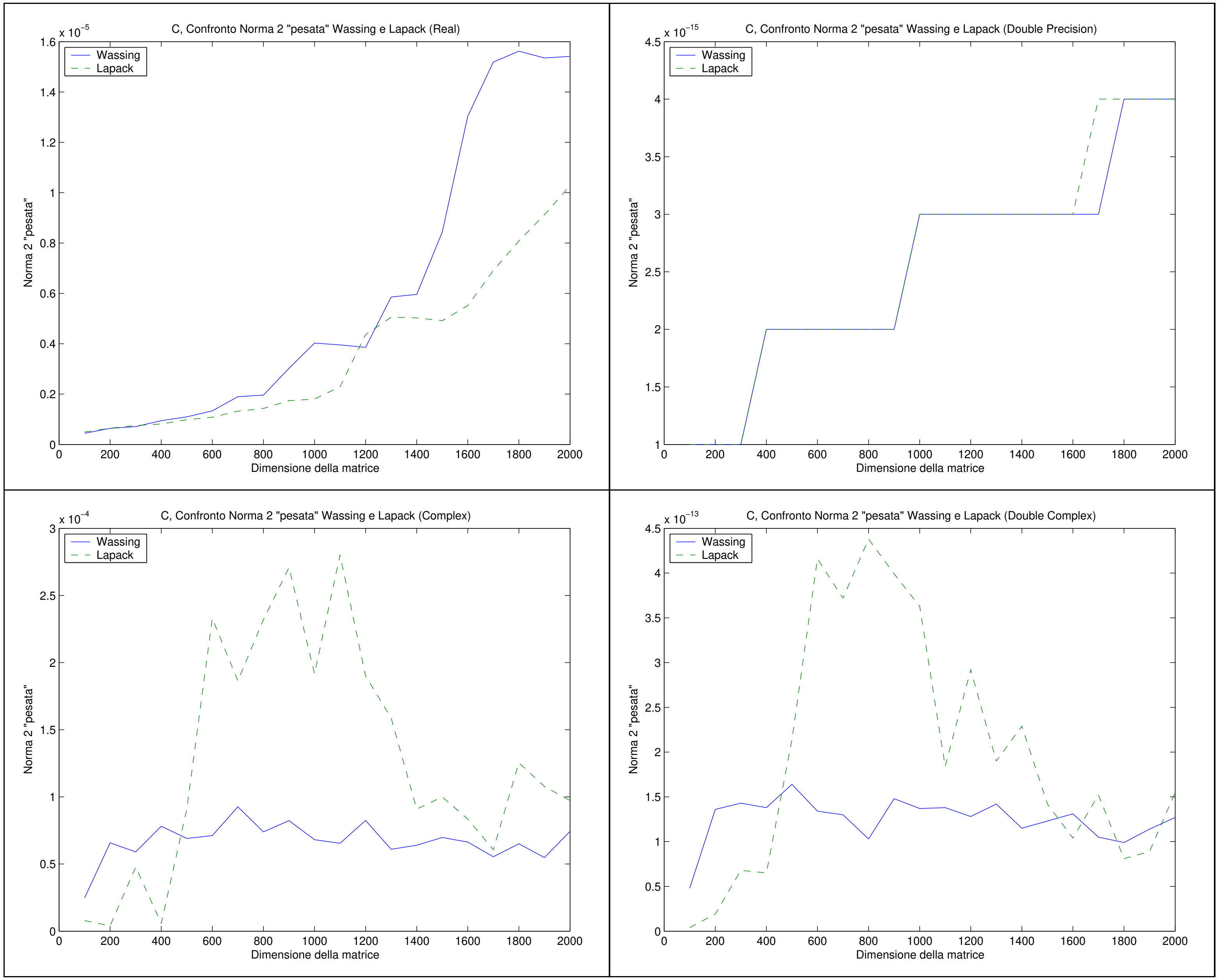

Tabella 4.17: Precisione, C, Confronto norma 2 “pesata” wasv e gesv

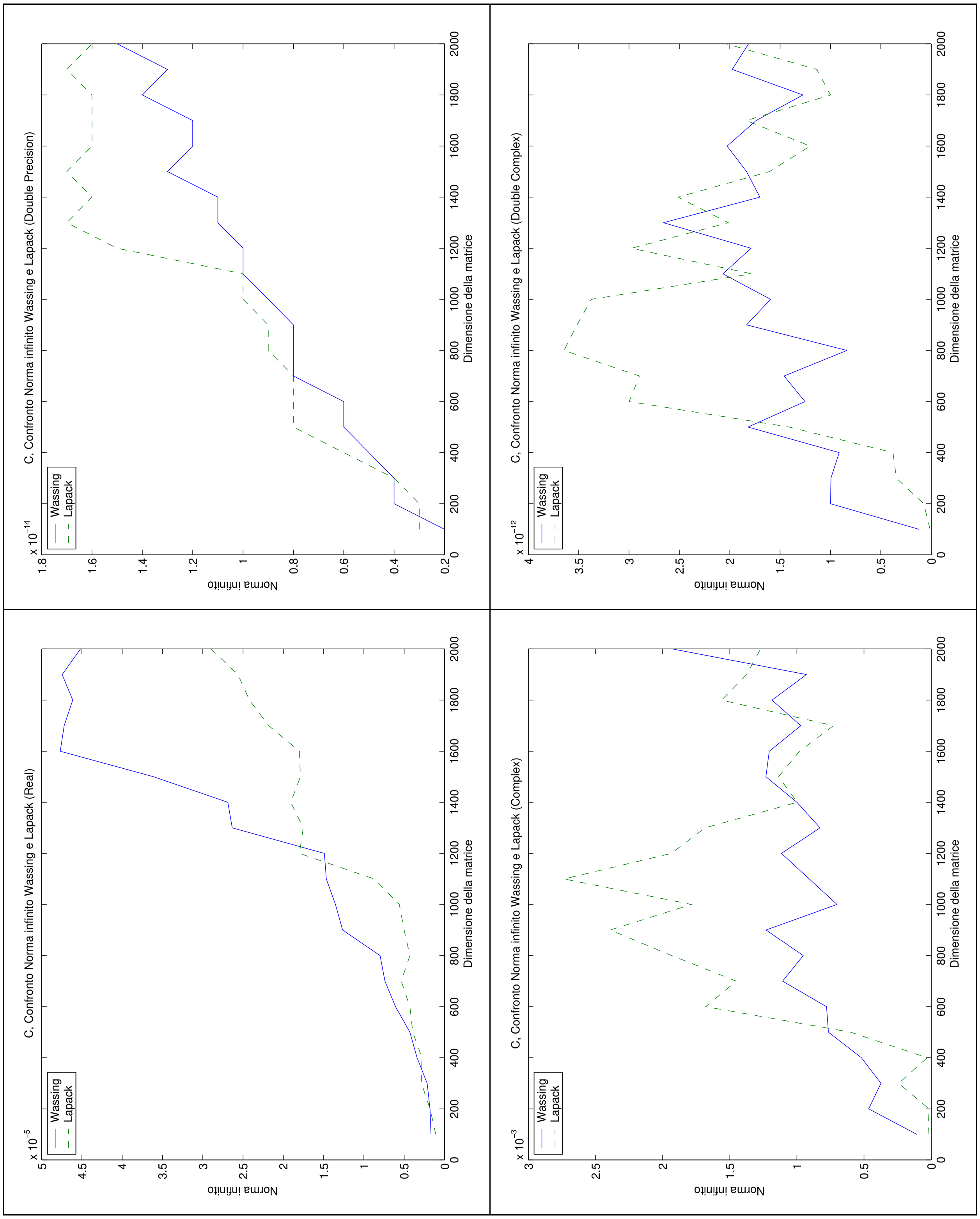

Tabella 4.18: Precisione, C, Confronto norma infinito `wasv` e `gesv`

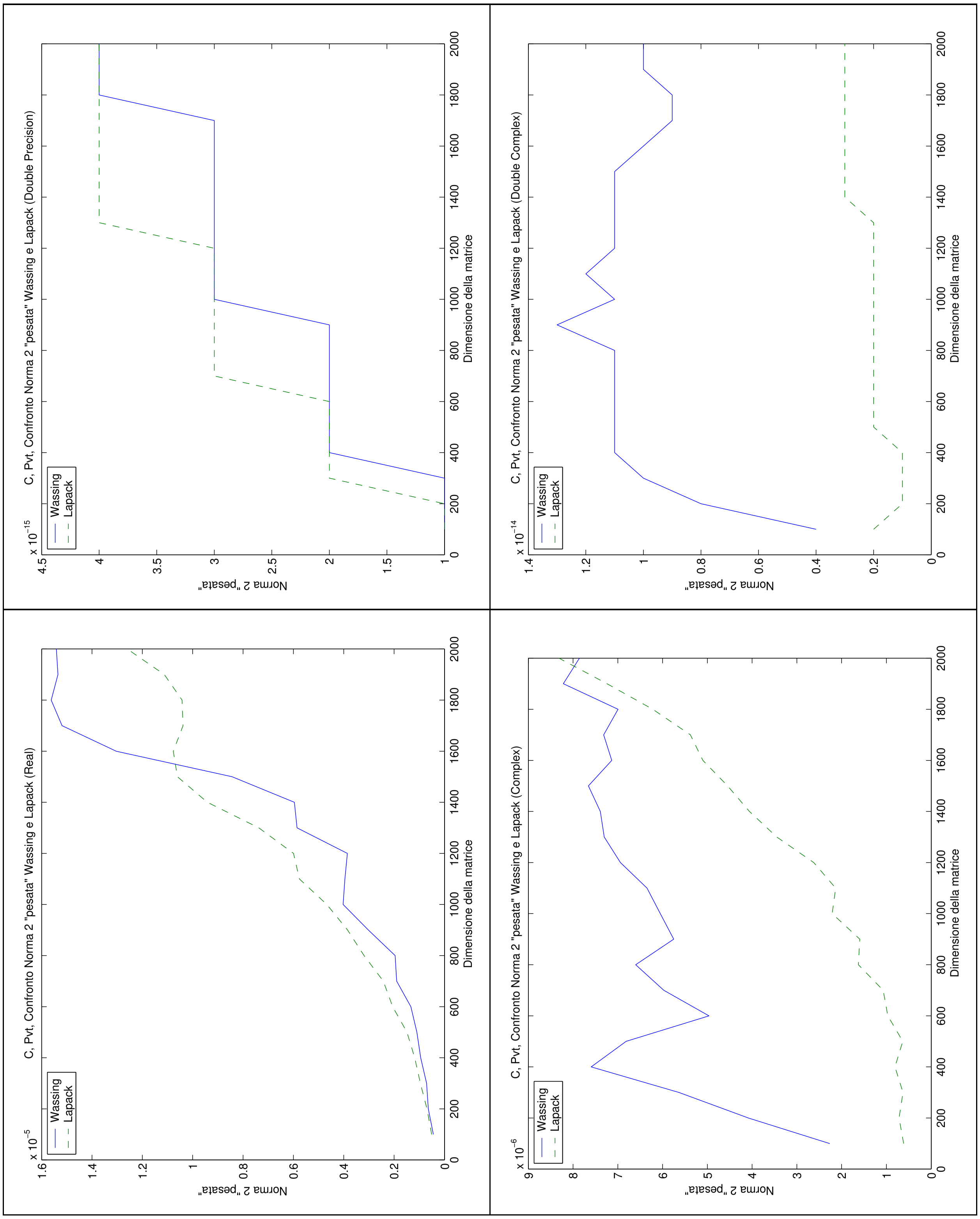

Tabella 4.19: Precisione, C, Confronto norma 2 “pesata” `wapvtsv` e `gesv`

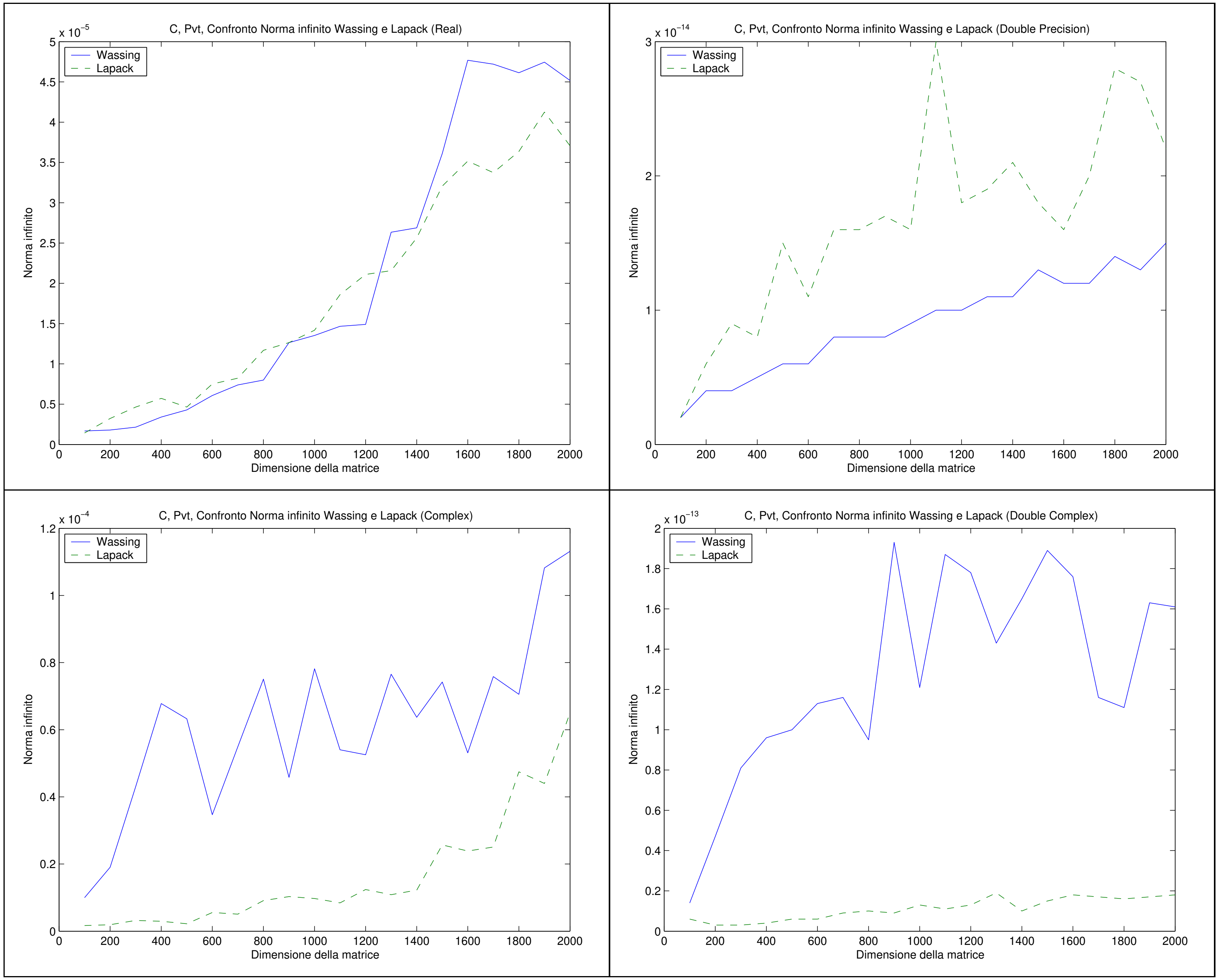

Tabella 4.20: Precisione, C, Confronto norma infinito `wapvtsv` e `gesv`

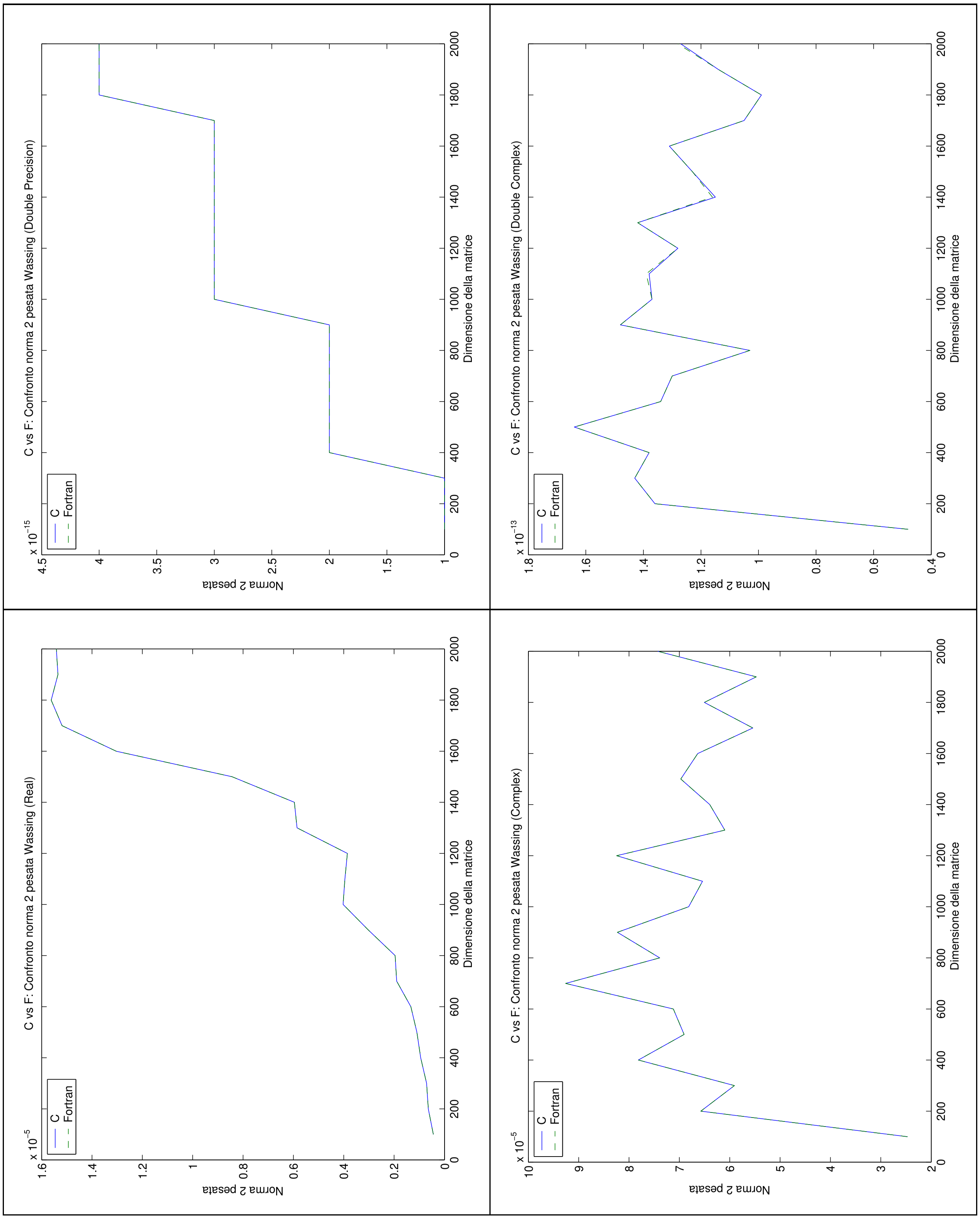

Tabella 4.21: Precisione, Confronto norma 2 “pesata” `wasv`, Fortran e C

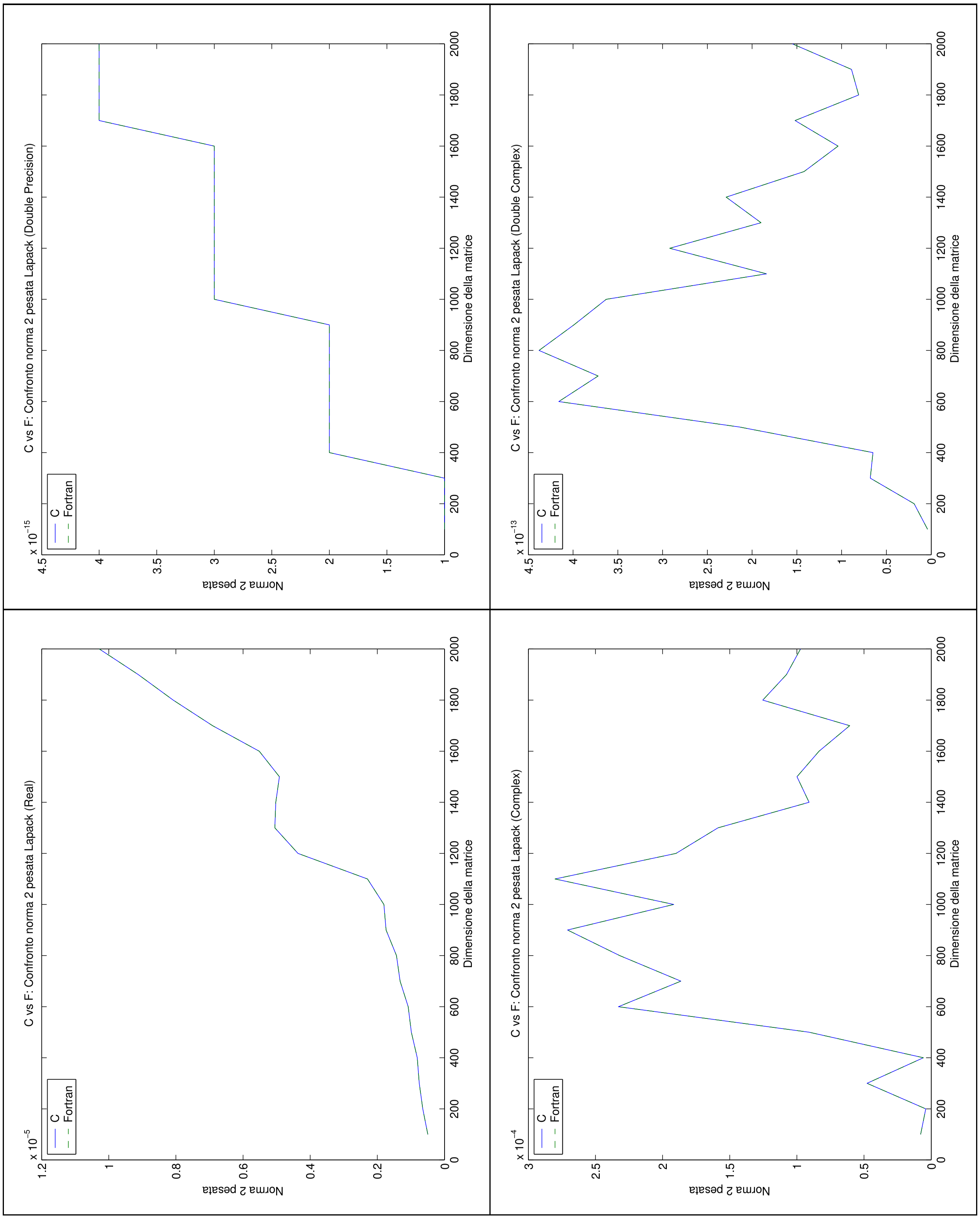

Tabella 4.22: Precisione, Confronto norma 2 “pesata” `gesv`, Fortran e C

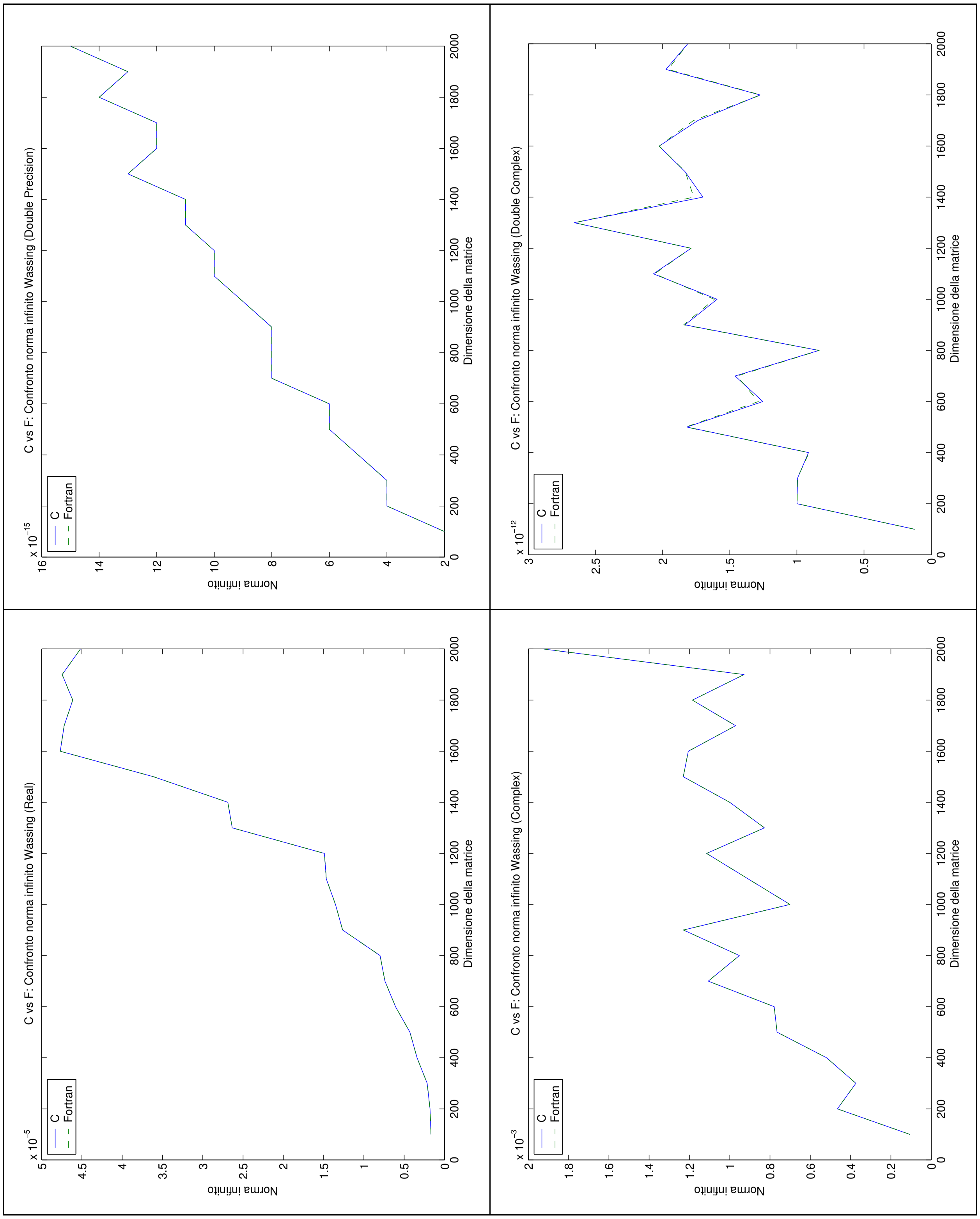

Tabella 4.23: Precisione, Confronto norma infinito `wasv`, Fortran e C

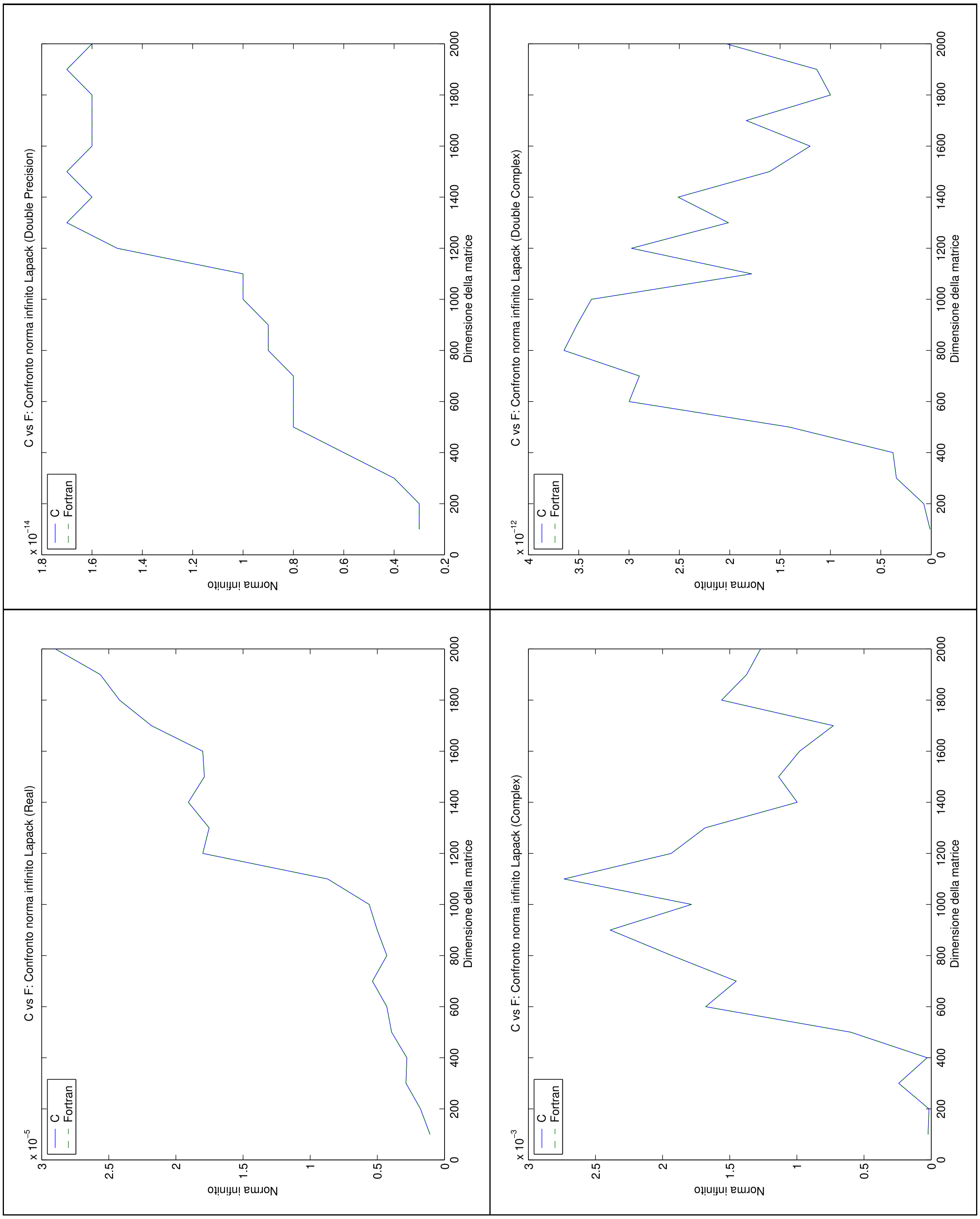

Tabella 4.24: Precisione, Confronto norma infinito `gesv`, Fortran e C

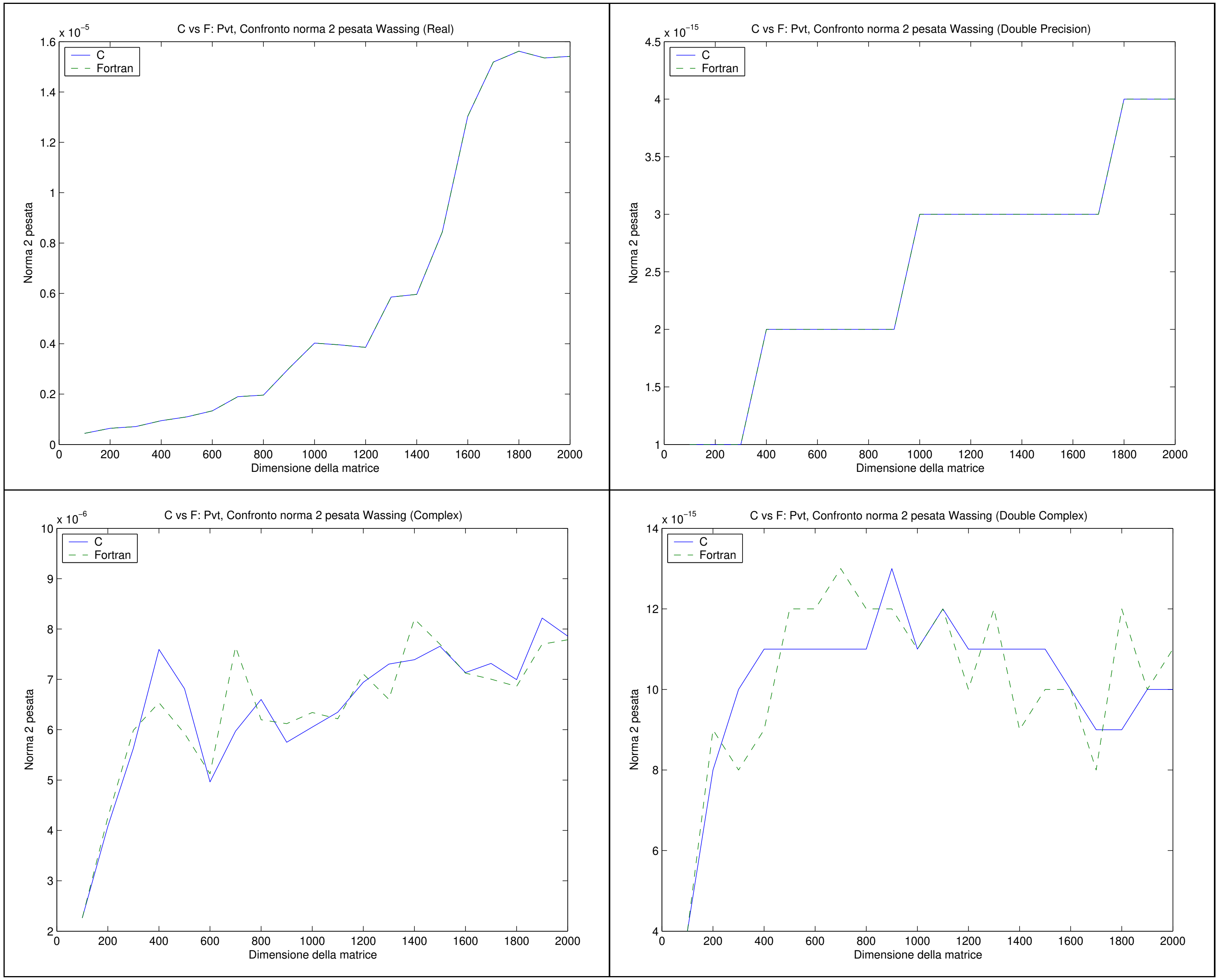

Tabella 4.25: Precisione, Pvt, Confronto norma 2 "pesata" `wasv`, Fortran e C

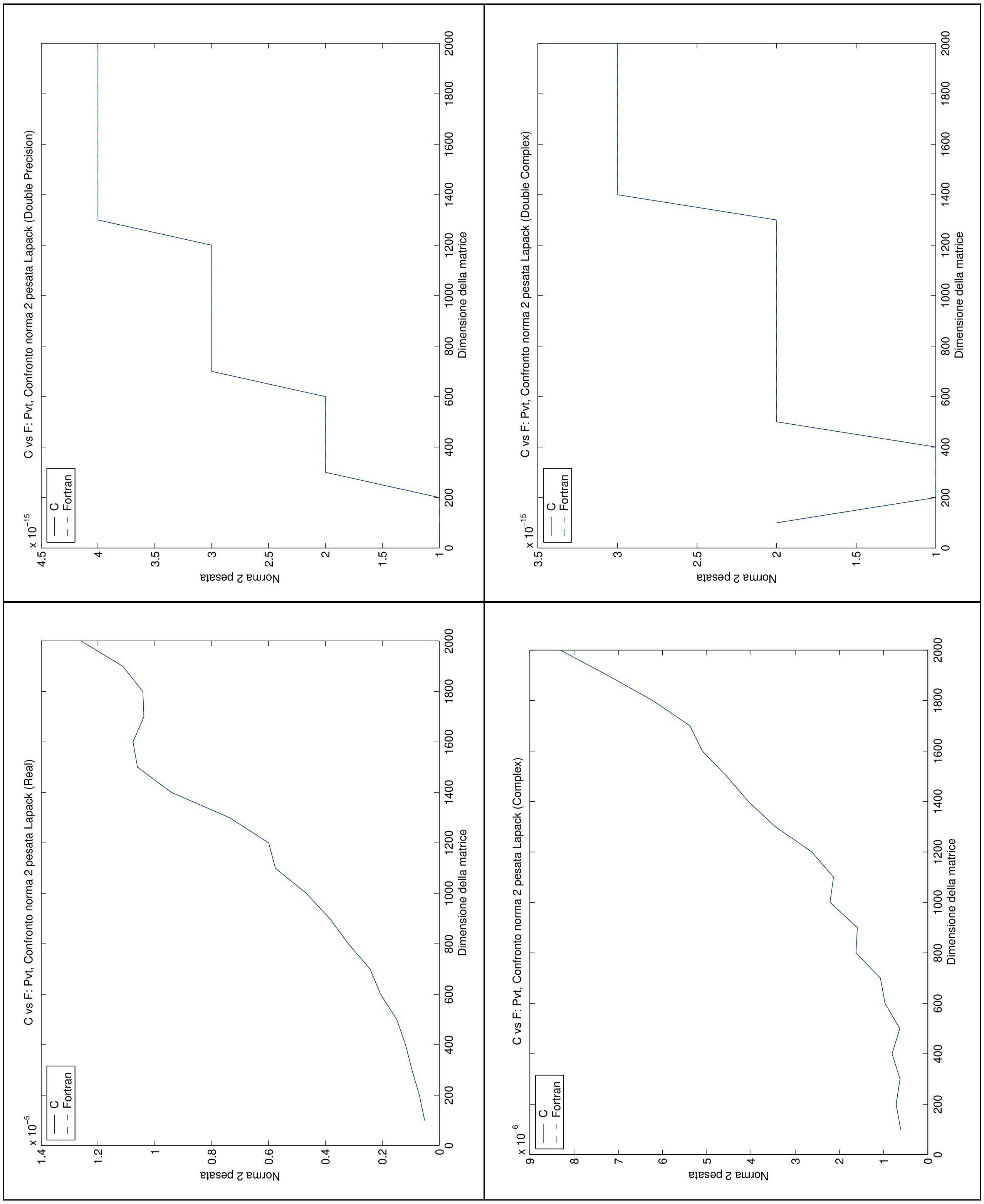

Tabella 4.26: Precisione, Pvt, Confronto norma 2 “pesata” `gesv`, Fortran e C

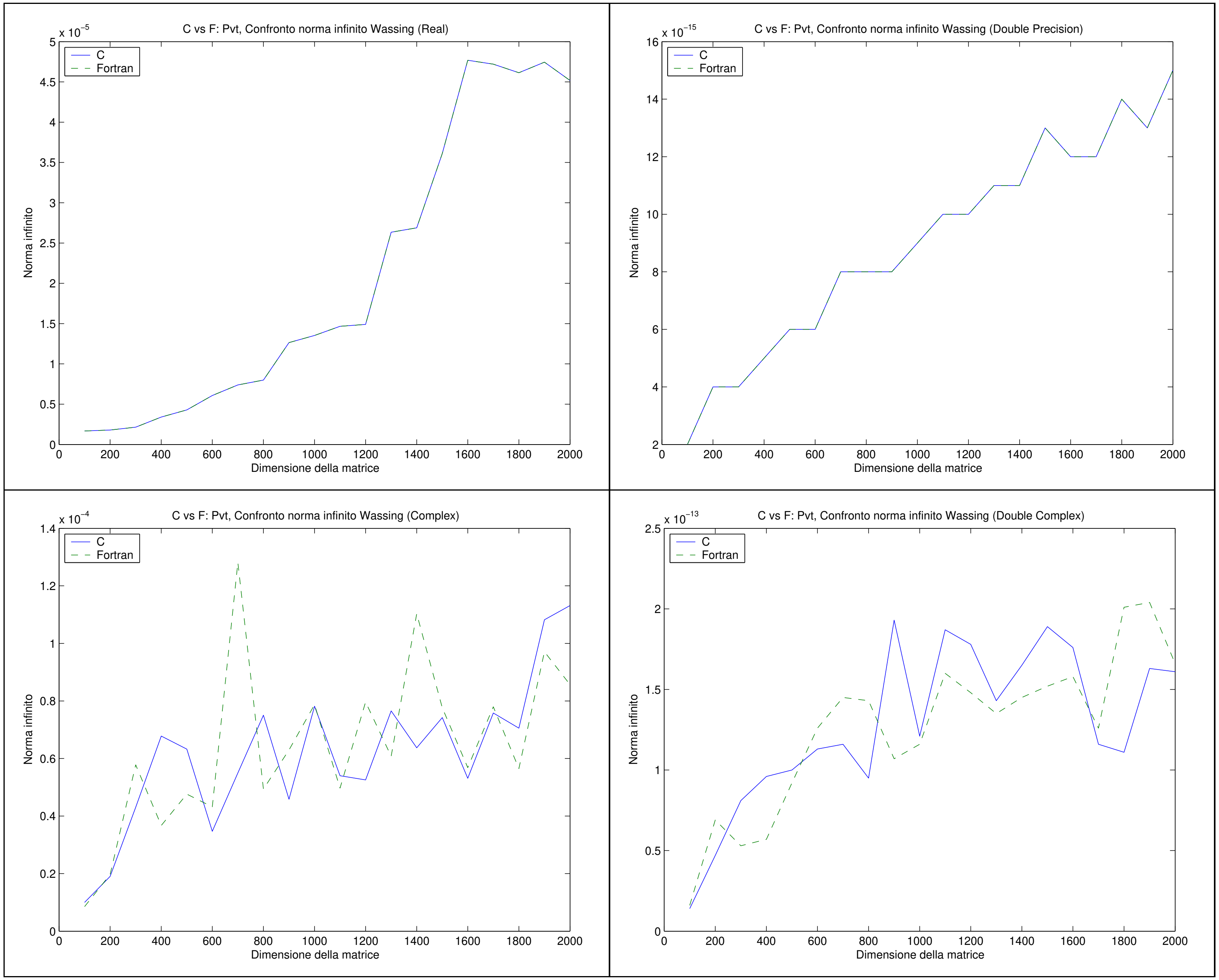

Tabella 4.27: Precisione, Pvt, Confronto norma infinito `wasv`, Fortran e C

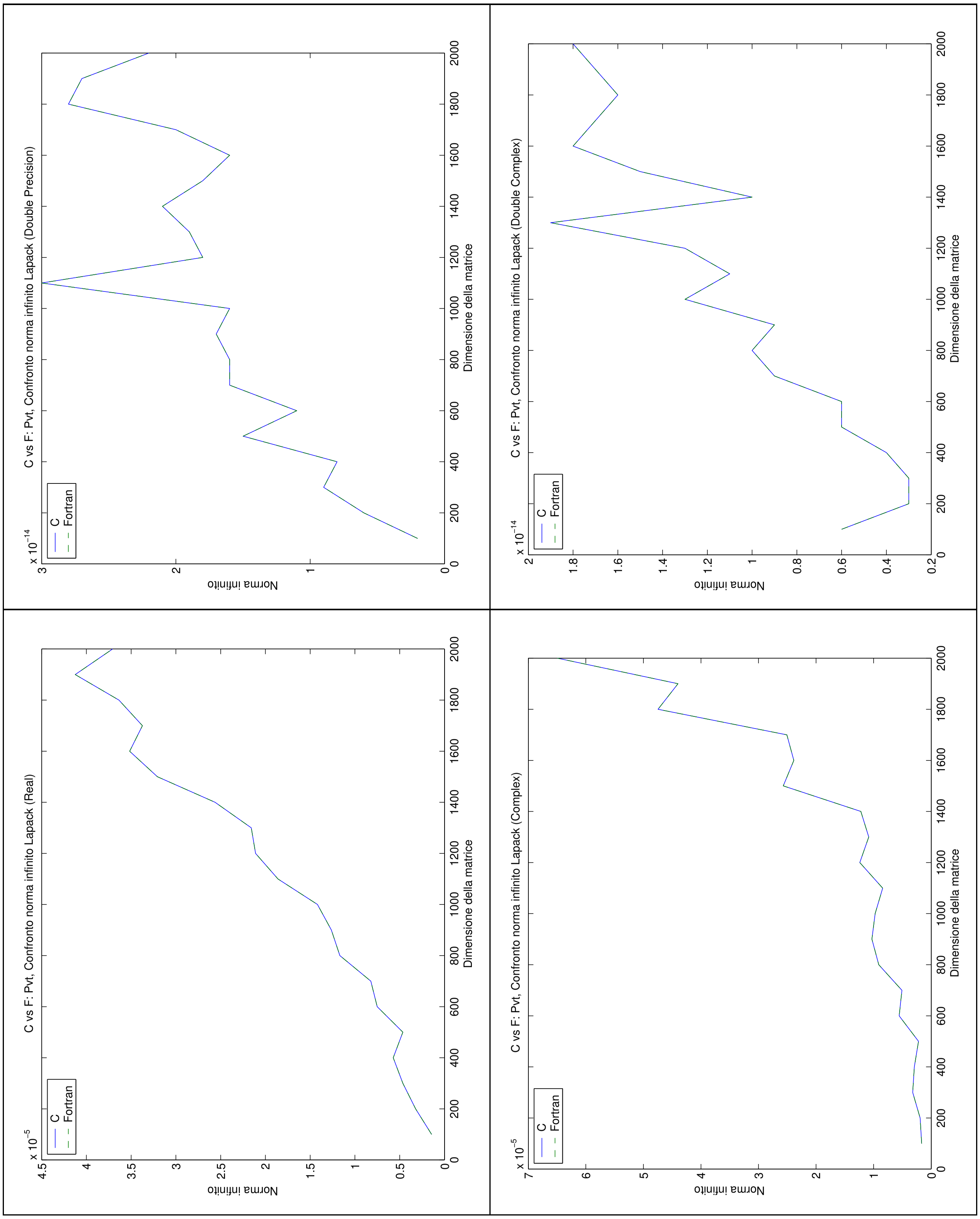

Tabella 4.28: Precisione, Pvt, Confronto norma infinito `gesv`, Fortran e C

Capitolo 5

Manuale utente

In questo capitolo si illustreranno le modalità di utilizzo dei sorgenti allegati; verranno descritte le routine principali in Fortran e C, e verranno mostrati alcuni esempi di compilazione dei programmi.

5.1 Nomenclatura

Nella scelta dei nomi da dare alle routine sviluppate, si è scelto di seguire lo schema già usato da Fortran e LAPACK:

- tutte le routine iniziano con la lettera corrispondente al tipo di dato che andranno ad usare:

  `s`: numeri reali in singola precisione;

  `d`: numeri reali in doppia precisione;

  `c`: numeri complessi in singola precisione;

  `z`: numeri complessi in doppia precisione.

- tutte le routine contengono due lettere, `wa`, che identificano l'implementazione del metodo di Wassing;
- le routine che implementano il pivoting parziale, contengono le lettere `pvt`;
- tutte le routine contengono, infine, le lettere `sv`, che indicano il *solving* di un sistema lineare.

Si può quindi generalizzare lo schema dei nomi in questo modo:

`{s|d|c|z}wa[pvt]sv`[1]

5.2 Funzioni per la generazione degli elementi di A

Come accennato in precedenza, il metodo di Wassing non richiede la memorizzazione esplicita della matrice. Risulta dunque necessario stabilire un formato standard per le funzioni che restituiscono gli elementi della matrice dei coefficienti del sistema lineare in esame.

Si tenga presente che la matrice è intesa essere A^T. Le funzioni, perciò, dovranno essere tali che `f(i,j)` $\equiv a_{ji}$, cioè richiedendo l'elemento (i, j) all'interno delle librerie per il metodo di Wassing, questo deve essere l'elemento (j, i) della matrice dei coefficienti del sistema lineare $Ax = b$. In seguito, si

[1]Con la notazione `{a1|...|an}` si intende che uno degli elementi `ai` deve essere indicato, mentre con `[x]` si indica la presenza opzionale di `x`; questa notazione viene utilizzata anche nel seguito del capitolo.

farà sempre riferimento ad A^T: l'elemento a_{ij} è l'elemento (i, j) di A^T, e dunque l'elemento (j, i) di A.

Il formato che deve essere rispettato da queste funzioni, è il seguente:

```
<tipo> <nome>(int i, int j, int n)
```

dove

`<tipo>`	è il tipo di dato opportuno
`<nome>`	è il nome scelto per la funzione
`i`	è la riga dell'elemento
`j`	è la colonna dell'elemento
`n`	è la dimensione della matrice

Le routine di risoluzione dispongono di un parametro, attraverso il quale viene indicato il `<nome>` della funzione da utilizzare per ottenere gli elementi della matrice dei coefficienti: in questo modo le routine sono più flessibili in quanto consentono di utilizzare un nome arbitrario per queste funzioni.

5.3 Routine di risoluzione in Fortran

Le routine Fortran restituiscono, in un parametro, il risultato dell'applicazione del metodo, come avviene nelle routine LAPACK; poi, a seconda che si utilizzi la versione con pivoting o meno, il numero ed il tipo dei parametri da passare cambia, come descritto di seguito:

`{s|d|c|z}wasv(n, b, nb, z, nz, c, getk, valid)`

calcola la soluzione di un sistema lineare $AX = B$, dove A è una matrice $\mathtt{n} \times \mathtt{n}$, e X e B sono matrici $\mathtt{n} \times \mathtt{nb}$, tramite il metodo di Wassing

`n`	Il numero di equazioni lineari, cioè, l'ordine della matrice A
`b`	In ingresso, la matrice B dei termini noti. In uscita, se `valid`$=0$, la matrice soluzione X
`nb`	Il numero delle colonne della matrice B
`z`	Vettore di appoggio per l'applicazione del metodo
`nz`	Dimensione del vettore `z`
`c`	Vettore di appoggio per contenere la colonna corrente
`getk`	Funzione per la restituzione degli elementi della matrice A
`valid`	Intero che indica il risultato della risoluzione: `valid`$=0$: risoluzione avvenuta con successo; `valid`$=k$: minore principale di ordine k nullo

`{s|d|c|z}wapvtsv(n, b, nb, z, nz, c, zt, ip, ipk, getk)`

calcola la soluzione di un sistema lineare $AX = B$, dove A è una matrice `n` $\times$ `n`, e X e B sono matrici `n` $\times$ `nb`, tramite il metodo di Wassing con l'utilizzo del pivoting parziale

`n`	Il numero di equazioni lineari, cioè, l'ordine della matrice A
`b`	In ingresso, la matrice B dei termini noti. In uscita, se `ip(n+1)`$=0$, la matrice soluzione X, con le righe permutate secondo il vettore `ip`
`nb`	Il numero delle colonne della matrice B
`z`	Vettore di appoggio per l'applicazione del metodo
`nz`	Dimensione del vettore `z`
`c`	Vettore di appoggio per contenere la colonna corrente

`zt` Vettore di appoggio per l'aggiornamento di `z`

`ip` Vettore di indici dei pivot che definisce la matrice di permutazione P; la riga i della matrice è stata scambiata con la riga `ip`(i);
`ip(n+1)`$= 0$: risoluzione avvenuta con successo;
`ip(n+1)`$= k$: minore principale di ordine k nullo

`ipk` Vettore di appoggio per l'applicazione della permutazione temporanea

`getk` Funzione per la restituzione degli elementi della matrice A

5.4 Routine di risoluzione in C

Nell'implementazione in C, le routine sono delle funzioni, il cui valore di ritorno indica se la risoluzione ha avuto successo o meno, in maniera simile a `valid` o `ip(n+1)` visto in precedenza per le routine Fortran.

```
int {s|d|c|z}wasv(int n, <tipo> b[], int nb, (* getk))
```

calcola la soluzione di un sistema lineare $AX = B$, dove A è una matrice `n` $\times$ `n`, e X e B sono matrici `n` $\times$ `nb`, tramite il metodo di Wassing

`n` Il numero di equazioni lineari, cioè, l'ordine della matrice A

`b` In ingresso, la matrice B dei termini noti. In uscita, se la matrice A è non singolare, la matrice soluzione X

`nb` Il numero delle colonne della matrice B

`getk` Funzione per la restituzione degli elementi della matrice A

`int {s|d|c|z}wapvtsv(int n, <tipo> b[], int nb, (* getk))`

calcola la soluzione di un sistema lineare $AX = B$, dove A è una matrice $\mathtt{n} \times \mathtt{n}$, e X e B sono matrici $\mathtt{n} \times \mathtt{nb}$, tramite il metodo di Wassing con l'utilizzo del pivoting parziale

- `n` Il numero di equazioni lineari, cioè, l'ordine della matrice A
- `b` In ingresso, la matrice B dei termini noti. In uscita, se la matrice A è non singolare, la matrice soluzione X, con le righe permutate secondo il vettore `ip`
- `nb` Il numero delle colonne della matrice B
- `getk` Funzione per la restituzione degli elementi della matrice A

5.5 Compilazione ed utilizzo

Di seguito, verrà mostrato come utilizzare i codici implementati, facendo uso di semplici esempi di compilazione.

5.5.1 Compilazione del codice Fortran

Per poter compilare il sorgente Fortran è necessario dichiarare, all'interno del programma principale, tutti i vettori di cui farà uso la routine di risoluzione. Definite, dunque, le strutture dati opportune, si devono anche dichiarare le funzioni per gli elementi della matrice dei coefficienti A e della matrice dei termini noti B; dopo aver costruito il vettore `b`, è infine possibile richiamare la routine di risoluzione.

Si consideri un codice di esempio:

```
Program sxolve

Implicit None

Integer n, nz, nb, i, j, k, valid

Parameter (n=10)
Parameter (nz=(n*n)/4)
Parameter (nb=1)

Real z(nz), c(n), b(n*nb)

Real sgetb, sgetk
External sgetb, sgetk

do i=1,n
   k=nb*(i-1)
   do j=1,nb
      b(k+j)=-sgetb(i,n)
   enddo
enddo

call swasv(n,b,nb,z,nz,c,sgetk,valid)

if (valid.ne.0) then
```

```
      print *, 'Minore principale di ordine ',valid,' nullo'
   endif


   end
```

Se il programma precedente viene chiamato `sxolve.f`, allora per compilarlo si utilizza la seguente linea di comando:

```
g77 -O3 -o <output> sxolve.f swasv.f sgetk.f sgetb.f
```

dove con `<output>` si è indicato il nome del file eseguibile risultato della compilazione. Sono state attivate anche le opzioni di ottimizzazione `-O3` che, come detto, consentono di ottenere eseguibili più performanti.

Nel caso il programma principale e le altre routine non si trovino tutti nella stessa directory, si dovrà precisare, oltre al nome del file, anche il percorso per raggiungerlo.

5.5.2 Compilazione del codice C

Per la compilazione del codice C le cose sono leggermente differenti da quanto visto in precedenza: per poter chiamare una funzione, è necessario dichiararne l'intestazione, che è composta dal tipo di ritorno, dal nome della funzione e dalla lista dei parametri con il rispettivo tipo. Per facilitare la fase di dichiarazione delle funzioni, è stato utilizzato il file `wassing.h` che contiene tutte le intestazioni delle funzioni utilizzate: esso deve essere incluso tramite la direttiva del preprocessore `#include wassing.h` all'interno del proprio codice sorgente. Dopo aver costruito il vettore dei termini noti `b`, è possibile richiamare la routine di risoluzione.

Si consideri il file sorgente che segue:

```
#include <stdio.h>
#include <stdlib.h>
#include "wassing.h"

#define dim 10
#define nbb 1

int main()
{
  float *b;
  int res, i, j;

  b=(float *) calloc(dim*nbb, sizeof(float));
  if (b==NULL) {
    fprintf(stderr, "%s: Impossibile allocare
                         la memoria necessaria\n", __FILE__);
    exit(1);
  }

  for (i=0; i<dim; i++)
  {
    for(j=0; j<nbb; j++)
      { b[nbb*i+j]=-sgetb(i+1,dim); }
  }
```

```
  res=swasv(dim, b, nbb, sgetk);

  if(res!=0)
  { printf("Minore principale di ordine %d nullo\n", res); }

  free(b);

  return 0;
}
```

che si suppone chiamato `sxolve.c`. Per compilarlo, si deve eseguire questa riga di compilazione:

```
gcc -O3 -o <output> sxolve.c swasv.c sgetk.c sgetb.c -lm
```

Come detto, il C non dispone dell'aritmetica dei numeri complessi: a questo proposito sono state definite delle macro del preprocessore, riunite nel file `ccomplex.h`, che deve essere incluso in maniera simile a `wassing.h` all'interno del codice che fa uso delle routine per i numeri complessi. Inoltre, deve anche essere indicato, sulla riga di compilazione, il file `ccomplex.c`: dunque, la riga di compilazione da eseguire sarà simile alla seguente (i nomi dei file iniziano con `c`, ad indicare l'utilizzo di numeri complessi in singola precisione, ed il programma principale è chiamato `cxolve.c`):

```
gcc -O3 -o <output> cxolve.c cwasv.c cgetk.c cgetb.c →
→ ccomplex.c -lm
```

È inoltre presente un file di intestazioni per le routine LAPACK, chiamato `lapack.h`, da usare come descritto in precedenza.

È necessario anche utilizzare il parametro di linking `-lm`, che include nel programma le funzioni matematiche della libreria C `math.h`, in quanto sia `wassing.h` che `ccomplex.h` le richiedono.

Anche in questo caso, è consigliabile attivare le ottimizzazioni.

Bibliografia

[1] AA.VV. *LAPACK – Linear Algebra PACKage. Version 3.0.* Disponibile presso `http://www.netlib.org/lapack/`.

[2] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*, third ed. SIAM, 1999. Disponibile presso `http://www.netlib.org/lapack/lug/index.html`.

[3] Baker, L. *C Tools for Scientists and Engineers.* McGraw-Hill, 1989.

[4] Cheney, W., and Kincaid, D. *Numerical Mathematics and Computing*, second ed. Brooks/Cole, 1985.

[5] Dahlquist, G., and Björck, Å. *Numerical Methods.* Prentice-Hall Series in Automatic Computation, 1974.

[6] Deuflhard, P., and Hohmann, A. *Numerical Analysis in Modern Scientific Computing: An Introduction*, second ed. Springer-Verlag, 2003.

[7] DOOLE, S., AND CHAMPNEYS, A. *Tacoma Narrows Bridge Disaster.* Disponibile presso `http://www.enm.bris.ac.uk/anm/tacoma/tacoma.html`.

[8] FORSYTHE, G. E., MALCOLM, M. A., AND MOLER, C. B. *Computer methods for mathematical computations.* Prentice-Hall Series in Automatic Computation, 1977.

[9] FOX, L. *An introduction to numerical linear algebra.* Oxford University Press, 1964.

[10] GOLUB, G. H., AND LOAN, C. F. V. *Matrix computations.* John Hopkins University Press, 1996.

[11] HALLIDAY, D., RESNICK, R., AND KRANE, K. S. *Fisica 2*, fourth ed. Casa Editrice Ambrosiana, 1994.

[12] HENDERSON, D. S., AND WASSING, A. A new method for the solution of $Ax = b$. *Numerische Mathematik*, 29 (1978), 287–289.

[13] HENRICI, P. *Essential of numerical analysis.* John Wiley & Sons, 1982.

[14] HULTQUIST, P. F. *Numerical Methods for Engineers and Computer Scientists.* Benjamin/Cummings, 1988.

[15] KAHANER, D., MOLER, C., AND NASH, S. *Numerical Methods and Software.* Prentice-Hall, 1989.

[16] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C programming language.* Prentice-Hall, 1978.

[17] Kernighan, B. W., and Ritchie, D. M. *The C programming language*, second ed. Prentice-Hall, 1989.

[18] Kronsjö, L. I. *Algorithms: their complexity and efficiency.* John Wiley & Sons, 1979.

[19] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C: the art of scientific programming*, second ed. Cambridge University Press, 1994.

[20] Rice, J. R. *Matrix computation and mathematical software.* McGraw-Hill, 1981.

[21] Smith, D. A Case Study and Analysis of the Tacoma Narrows Bridge Failure. Engineering Project 99.497, Department of Mechanical Engineering, Carleton University, Ottawa, Canada, March 29, 1974. Supervised by Professor G. Kardos.

[22] Stoer, J., and Bulirsh, R. *Introduction to Numerical Analysis*, second ed. Springer-Verlag, 1992.

[23] Wassing, A. Solving $Ax = b$: a method with reduced storage requirements. *SIAM J. Numer. Anal. 19*, 1 (February 1982).

[24] Wilkinson, J. H. Error analysis of direct methods of matrices inversion. *J. Assoc. Comp. Mach.*, 8 (1961), 281–300.

[25] Wilkinson, J. H. *Rounding Errors in Algebraic Processes.* Prentice-Hall, 1963.

[26] Wilkinson, J. H. *The algebraic eigenvalue problem.* Oxford University Press, 1988.