Introduzione alla programmazione in FORTRAN

Università degli Studi di Padova
Corso di Calcolo Numerico per Ingegneria Meccanica - Sede di Vicenza

a.a.2005/2006

DISPENSA

Programmare in Fortran

Questa dispensa vuole essere una semplice introduzione al Fortran con lo scopo di aiutare lo studente a scrivere ed eseguire dei programmi che risolvano problemi di Calcolo Numerico.

Sommario

1 Introduzione

Il linguaggio di programmazione Fortran (FORmula TRANslation) è stato pensato e progettato in particolare modo per gli ingegneri e gli scienzati. Da oltre 40 anni

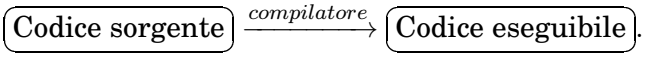
il Fortran viene utilizzato per progettare la struttura di aereoplani, per analizzare dati, per predire il flusso di contaminanti radioattivi in mezzi porosi... per risolvere, cioè, problemi che possono essere descritti in termini matematici e risolti mediante un'approssimazione numerica.

Il Fortran può essere definito come un *linguaggio di programmazione di alto livello*: i programmi, infatti, non vengono scritti direttamente nel linguaggio di macchina ma, al contrario, in un linguaggio che è ben comprensibile per chi scrive il programma stesso.

L'utente scrive il *codice sorgente*, cioè un *algoritmo* capace di risolvere un determinato tipo di problema, facendo uso di una serie di *comandi* che definiscono operazioni elementari con i dati del problema stesso.

Il codice sorgente, tuttavia, per poter essere *eseguibile* al computer, cioè capace di leggere i dati di input, eseguire l'algoritmo di risoluzione del problema e dare il risultato, deve essere *compilato*: la compilazione del codice è effettuata da un programma separato, detto *compilatore*, che traduce il codice sorgente in un codice eseguibile scritto nel linguaggio di macchina.

Si ha perciò:



La fase di creazione del codice sorgente è dunque la fase più importante della programmazione in Fortran, in quanto il codice eseguibile non fa altro che eseguire i comandi dati nel codice sorgente e opportunamente tradotti nel linguaggio di macchina.

Occorre dunque:

- ✔ **specificare il problema** che si vuole risolvere con il programma;
- ✔ **analizzare il problema**: prima di mettersi a scrivere nel linguaggio di programmazione è fondamentale avere chiaro l'algoritmo che risolve il problema stesso, quali sono i dati da dare in input, quale l'output; controllare le dimensioni fisiche dei dati... prima di mettersi a lavorare al computer, quindi, è bene scrivere su un foglio di carta ciò che si vuole fare.
- ✔ **scrivere il codice sorgente**: una volta analizzato il problema, si può scrivere il programma, tenendo presente che in Fortran alcune funzioni sono *intrinseche* come la radice quadrata, la funzione seno, coseno, logaritmo, esponenziale, e possono essere usate direttamente. Funzioni più complicate, invece, devono essere scritte come un sottoprogramma e richiamate nel programma principale quando servono.
- ✔ **compilare il codice**, cioè tradurre il programma nel linguaggio di macchina. Si ottiene, in tal modo, un programma eseguibile.
- ✔ **eseguire e testare il programma**. Il programma deve essere scritto bene non solo nella sintassi del linguaggio di programmazione ma anche nella

descrizione dell'algoritmo stesso. Fintantochè il programma non è ben scritto sintatticamente, nella fase di compilazione non si ottiene l'eseguibile ma la lista di tutti gli errori... Ma anche quando il codice è eseguibile potrebbero esserci degli errori non di sintassi ma di descrizione dell'algoritmo stesso. Perciò è bene testare il programma con problemi di cui si conosce la soluzione. Per esempio, se il codice deve calcolare gli zeri di una funzione, prima di cercare gli zeri di funzioni complicatissime, è bene testare il codice con funzioni più semplici di cui si conoscono analiticamente gli zeri.

2 Strumenti di lavoro

- **L'editor per scrivere il codice sorgente** Per scrivere il codice sorgente, occorre avere a disposizione un *text editor* cioè un editore di testo - quali, ad esempio, `Notepad` su Windows, o `emacs` su ambienti Windows e Linux.

  Il codice sorgente sarà chiamato con un nome dato dall'utente a cui si aggiunge l'estensione `.f` (a significare che è il codice sorgente di un programma Fortran). Per esempio, se vogliamo scrivere un codice per calcolare gli zeri di una funzione e lo vogliamo chiamare "zeri", andremo a scrivere un codice sorgente dal nome `zeri.f`.

- **La compilazione del codice** Per quanto riguarda la compilazione del codice sorgente, ve ne sono diversi sia in commercio sia distribuiti gratuitamente via rete che variano a seconda della versione di Fortran utilizzata per programmare.

  Noi ci soffermiamo sul Fortran nella versione nota come Fortran 77 che risale al 1978. Versioni più recenti sono il Fortran 90 (del 1991), il Fortran 95 (del 1996) e il Fortran 2003 (recentissimo), ciascuna delle quali ingloba le versioni standard precenti e vi apporta modifiche e miglioramenti.

  Uno dei compilatori del Fortran 77 è il `g77` (GNU Fortran).

- **Il terminale** Per compilare il codice e per fare eseguire il codice stesso, i comandi e le informazioni sono dati attraverso un *command-line terminal*: attraverso una finestra di `terminal` (o `shell`) digitiamo i comandi che interessano.

  Ad esempio, una volta scritto il codice `zeri.f` e utilizzando il compilatore `g77`, per rendere eseguibile il codice scriveremo il comando:

  ```
  g77 zeri.f -o zeri
  ```

  facendo attenzione che il file `zeri.f` si trovi nella directory (cartella) in cui abbiamo scritto questa riga di comando.

  Questo comando fa sì che il compilatore `g77` legga il codice sorgente `zeri.f`, controlli che non vi siano errori nel file e traduca il codice nel linguaggio di macchina producendo il file eseguibile dal nome `zeri`.

Se ci sono errori, allora il compilatore li segnala con dei messaggi che vengono visualizzati sul terminale stesso e non produce il file eseguibile. Infatti, in tal caso, bisogna correggere il codice sorgente, salvare le correzioni fatte e digitare nuovamente il comando di compilazione.

Se gli errori sono stati tutti corretti avremo l'eseguibile, altrimenti saranno visualizzati nuovamente i messaggi di errori... Il codice eseguibile, infatti, non viene prodotto fintantochè il codice sorgente non è completamente corretto sintatticamente.

Quando finalmente non ci saranno errori, avremo il codice eseguibile `zeri` - basta fare una verifica controllando la lista dei files presenti nella directory in cui stiamo lavorando.

Potremo quindi far girare il codice. Digitando sulla shell la linea di comando

```
zeri
```

faremo girare il codice e avremo il risultato o sul terminale stesso o su un file di dati, a seconda di come sia stato scritto il codice stesso (osserviamo che i dati di input sono immessi o tramite terminale o mediante un file di dati, come vedremo in seguito).

3 Prime regole del Fortran 77

Ogni programma scritto in Fortran 77 ha delle precise regole di *column indentation* (rientranza sulle colonne), per cui il testo del codice sorgente non può essere scritto *ovunque si voglia* nel *buffer* dell' *editor*!

- **Colonna 1**: è riservata ai commenti. Se uno pone nella prima colonna la lettera `C` (per commento), o `*`, `!` o qualunque altro carattere, allora il resto della linea **viene ignorato** nella compilazione del codice. Perciò questa linea è utile per scrivere dei commenti e delle spiegazioni sul codice stesso (commenti e spiegazioni che tornano utili quando riprendiamo il codice dopo che è passato un po' di tempo o se vogliamo far lavorare un'altra persona con quel codice stesso).

  Osserviamo che un commento può essere scritto anche dopo una linea che contiene delle istruzioni.

  Esempi

```
!  Questa è un'intera linea di commento

      ........
   do i=1,n   !questo ciclo viene ripetuto n volte
      ........
```

- **Colonne 2 - 5**: queste colonne sono riservate per mettervi delle *label* numeriche, cioè dei numeri che servono per identificare delle istruzioni che non vengono eseguite in maniera sequenziale, oppure per definire un formato di scrittura/lettura di alcune variabili (tutti concetti che vedremo più avanti).

- **Colonna 6**: se un'istruzione è troppo lunga e si deve andare a capo, allora nella colonna 6 si pone un carattere (ad esempio +), per indicare che su quella linea continua l'istruzione della linea precedente. Se si deve andare nuovamente a capo, si pone un'altro carattere (può essere lo stesso di prima o uno diverso) e si continua a scrivere l'istruzione.

  Esempio

  Si vuole scrivere l'istruzione che assegna alla variabile `det` il determinante di una matrice 3×3 i cui elementi a_{ij}, $i = 1, 2, 3$, $j = 1, 2, 3$ sono stati memorizzati nelle variabili `a11, a12, a13, a21, a22, a23, a31, a32, a33`. L'espressione del determinante (calcolandolo rispetto alla prima riga) possiamo scriverla su più linee come:

```
      det=a11*(a22*a33-a23*a32)+
     +      a12*(a23*a31-a21*a33)+
     +      a13*(a21*a32-a22*a31)
```

- **Colonne 7-72**: su queste colonne si scrivono tutte le istruzioni del codice sorgente[1] Se si deve andare oltre la 72-sima colonna allora si deve andare a capo ponendo un carattere sulla colonna 6 della riga successiva.

Altre osservazioni utili per la programmazione:

- ✗ il codice deve essere ben leggibile non solo da chi lo ha scritto ma anche da altri utenti. Di qui l'importanza dei commenti e di scrivere le strutture `do`, `if`, `do while` con le opportune rientranze (come vedremo in seguito).

- ✗ Il Fortran 77 non fa distizioni tra caratteri maiuscoli e minuscoli.

4 Programma d'esempio e istruzioni fondamentali

Scriviamo ora il nostro primo programma in Fortran, che sia in grado di fare il prodotto di due numeri reali.

[1] Perchè ci sono queste limitazioni sulle colonne? Quando il Fortran è nato, sul finire degli anni cinquanta, i computers erano ancora agli inizi della loro evoluzione e la fase finale di un codice consisteva nel perforare una scheda che conteneva istruzioni binarie. La scheda standard aveva 80 colonne, numerate da 1 a 80 da sinistra a destra.

Queste schede ebbero vita agli inizi degli anni trenta, e furono simbolo di modernità; negli anni sessanta ebbero il loro fulgore. Oggi sono ormai dimenticate e fanno parte della storia del computer.

I dati di input sono dunque `a` e `b`. Il dato di output è il loro prodotto, che memorizziamo come variabile `c`.

```
12---67------------------------------------------------------------72

      program prodotto

C  definizione delle variabili
C
      implicit none
      real a,b,c

      write(6,*) 'primo numero di cui fare il prodotto'
      read(5,*) a
      write(6,*) 'secondo numero da moltiplicare al primo'
      read(5,*) b

      c=a*b
      write(6,*) 'Il prodotto di ', a, ' e ', b, ' e'': ', c

      end
```

Spieghiamo le istruzioni di questo semplice programma:

- **Inizio e fine del programma**

  Il programma **inizia** con l'istruzione

  `program NOMEPROGRAMMA`

  e **termina** con l'istruzione

  `end`.

- **Commenti al programma**

  Ci sono delle righe di commento (anche se questo programma non richiede molti commenti su ciò che deve fare).

- `implicit none`

  L'istruzione `implicit none` dice che tutte le variabili sono dichiarate. In questo caso, le variabili `a, b, c` sono dichiarate come variabili reali.

  Dichiarare tutte le variabili evita molti errori: se non fosse posta l'istruzione `implicit none` e se una variabile fosse dichiarata come reale, ma poi venisse scritta in modo errato in una istruzione, nel compilare il codice non verrebbe segnalato alcun messaggio di errore perchè essa sarebbe automaticamente dichiarata, in qualche modo, dal compilatore stesso, ma il codice eseguibile

potrebbe dare risultati errati perchè quella variabile non è stata dichiarata nel modo corretto (ad esempio, una variabile reale diventa intera). Il programma darà quindi dei risultati ma saranno errati. Conviene perciò scrivere sempre l'istruzione `implicit none`.

Esempio

Si compili il codice `prodotto.f` ottenendo l'eseguibile `prodotto`:

```
g77 prodotto.f -o prodotto
```

Si faccia girare il codice per calcolare il prodotto di `a=1.5` e `b=3.5`. Lanciando il comando `prodotto` sulla shell si leggerà la riga

```
primo numero di cui fare il prodotto
```

e noi scriveremo 1.5 e clickeremo sul tasto `Enter`.

Comparirà quindi sulla shell la riga

```
secondo numero da moltiplicare al primo
```

e noi digiteremo 3.5 e clickeremo il tasto di Invio.

Leggeremo sulla shell il messaggio

```
Il prodotto di 1.5 e 3.5 e':  5.25.
```

Proviamo ora a commentare nel nostro codice `prodotto.f` l'istruzione di `implicit none` e anzichè scrivere `c=a*b`, scriviamo `i=a*b`. Analogamente nell'istruzione di `write` scriviamo `i` al posto di `c`. Ma nella dichiarazione delle variabili non dichiariamo la `i`. Salviamo questo nuovo codice, compiliamo ed eseguiamolo per calcolare lo stesso prodotto. Questa volta vedremo sulla shell la scritta:

```
Il prodotto di 1.5 e 3.5 e':  5
```

Abbiamo ottenuto, quindi, un risultato sbagliato.

Perciò conviene **sempre dichiarare tutte le variabili** utilizzando il comando `implicit none`: se dimentichiamo di dichiarare una variabile, nel momento della compilazione saranno segnalate tutte le variabili non dichiarate!

✚ Dichiarazione delle variabili

Il passo successivo è quindi quello di dichiarare tutte le variabili che potranno essere di tipo intero (`integer`), reale (`real`), logico (`logical`), di caratteri alfanumerici (`character`).

Le variabili numeriche sono date in input e in output in formato decimale ma sono memorizzate in formato binario. Perciò non sempre si ha esatta corrispondenza tra il numero dato in input e lo stesso numero letto in Fortran: basti tenere presente il fatto che molti numeri reali hanno una rappresentazione finita in formato decimale ma infinita in formato binario (per cui facendo poi la conversione da numero binario a numero decimale non si ha la corrispondenza iniziale).

Esempio

Se lanciamo il codice `prodotto` dove `a` = 1.1 e `b`= 2.4 otterremo il risultato:

```
Il prodotto di 1.10000002 e 2.4000001 e':  2.6400001.
```

Il problema della rappresentazione dei numeri reali è uno dei più delicati nella risoluzione dei problemi numerici. Proprio per questo un numero reale può essere dichiarato in singola precisione o in doppia precisione: la variabile viene memorizzata occupando più o meno spazio di memoria (in termini di bit, più o meno bit) e, di conseguenza, maggiore o minore precisione.

Esempio

Se le variabili `a, b, c` sono dichiarate in doppia precisione come `real*8`, con `a` = 1.1 e `b`= 2.4 avremo il risultato:

```
Il prodotto di 1.1 e 2.4 e':  2.64
```

In formato binario i numeri 1.1 e 2.4 continuano ad avere una rappresentazione infinita ma la conversione, in formato decimale, di un numero memorizzato in doppia precisione permette una rappresentazione corretta.

Una variabile è, dunque, rappresentata da un nome simbolico (nel nostro caso abbiamo `a` o `b` o `c`) e ha a disposizione un certo spazio nella memoria del computer (che può cambiare da computer a computer). Il *valore* della variabile può cambiare all'interno del codice se viene assegnato ad esso un nuovo valore, ma *il tipo* di variabile non cambia perchè rimane quello definito nella dichiarazione delle variabili.

Se, ad esempio, oltre ad `a, b, c`, nel codice dobbiamo lavorare con delle variabili intere `i, j`, dobbiamo dichiarare queste ultime come intere:

```
integer i,j
real a,b,c
```

Quindi `i, j` sono dei nomi simbolici di variabili intere, cioè potranno assumere solo valori di numeri interi, mentre `a, b, c` sono dei nomi simbolici di variabili reali, cioè potranno assumere solo valori reali.

Oltre alla rappresentazione di numeri interi o reali (scalari) è possibile, in Fortran, lavorare anche con vettori e matrici, come vedremo in seguito.

Qui diamo un prospetto dei tipi di variabili comunemente utilizzati in Fortran:

nome simbolico	locazione di memoria	descrizione
`logical`	1 bit	assume i valori `.true.` (1) o `.false.` (0)
`integer`	4 bytes	numeri interi
`real`	4 bytes	numeri reali in singola precisione
`double precision`	8 bytes	numeri reali in doppia precisione
`real*n`	n bytes	numeri reali in precisione singola (`n=4`) doppia (`n=8`) o a 16 bytes (`n=16`)
`complex`	2×4 bytes	due numeri `real`: parte reale e immaginaria di un complesso
`double complex`	2×8 bytes	due numeri `double precision` : parte reale e immaginaria di un complesso
`character`	1 byte	un carattere alfanumerico
`character*n`	n bytes	n caratteri alfanumerici

Le variabili, siano esse intere o reali, non possono assumere infiniti valori ma possono variare in un intervallo limitato, che varia a seconda della macchina utilizzata. Ad esempio, una variabile `integer` può variare nell'intervallo $[-2^{31}, 2^{31}-1]$, mentre una variabile `real` può variare in un intervallo che varia da circa 10^{-38} a 10^{+38}, permettendo una precisione di 7 cifre decimali.

Si hanno errori di *underflow* o *overflow* quando si vuole rappresentare un numero che non appartiene all'intervallo rappresentabile - chiamando con min il minimo dell'intervallo di rappresentabilità e con max il massimo di tale intervallo, si ha *underflow* quando si vuole rappresentare un valore x tale che $-min < x < +min$. Si ha invece *overflow* quando $|x| > max$.

✚ Una modalità di inserimento dei dati di input

Passiamo ora a descrivere le righe del programma relative alle istruzioni di `write` e `read`.

Un'istruzione del tipo

```
write(6,*) 'blablabla'
```

rende disponibile all'esterno l'informazione contenuta tra apici, in tal caso, la scritta `blablabla`.

Il numero 6 è associato al terminale, cioè la visualizzazione avviene sulla shell dove viene lanciato il codice. Al posto di 6 potremmo mettere `*` e le cose non cambierebbero. Se invece di volere una visualizzazione su terminale, volessimo scrivere `blablabla` su un file, dovremmo mettere al posto di 6 un altro numero da associare ad un appropriato file, come vedremo in seguito.

L'istruzione del tipo

```
read(5,*) a
```

impone invece all'elaboratore di leggere *in qualche modo* un valore e di memorizzarlo nella variabile `a`. In quale modo? Il numero 5 indica che la lettura avviene attraverso il terminale cioè chi lancia il codice dovrà manualmente

scrivere il valore che si vuole assegnare ad `a`. Notiamo che, nell'eseguire il codice, se non ci fosse l'indicazione data dal comando

```
write(6,*) primo numero di cui fare il prodotto
```

noi non capiremmo che dobbiamo scrivere il valore del numero e clickare poi il tasto di Invio, in quanto la sola istruzione di `read(5,*)` lascia l'elaboratore in attesa del dato e, fino a quando non verrà introdotto dall'utente, non verrà fatto niente. Questo modo di dare i dati di input non è molto conveniente, sia perchè occorre scrivere prima di ogni `read` una istruzione di `write` sia perchè è facile sbagliare il dato di input ed essere costretti a far girare di nuovo il codice per inserire correttamente tutti i dati di input. Inoltre, vi possono essere troppi dati di input che rendono improponibile questa modalità di inserimento dei dati. Il miglior modo di inserire i dati è quello di scriverli su di un file separato rispetto al programma Fortran e far leggere dal programma i dati contenuti nel file. Questo file di dati di input sarà etichettato con un numero particolare che verrà richiamato dal comando di `read`.

Le operazioni di input e output le vedremo meglio nel seguito. Qui diamo ulteriori spiegazioni sull'asterisco che troviamo sia nel comando di `write` sia nel comando di `read`: si riferisce al tipo di formato (o stile) in cui scrivere o leggere i dati (siano essi numeri o caratteri alfanumerici). Il semplice `*` è il formato di *default*, ma può essere cambiato dall'utente. Anche questo lo vedremo nel seguito.

✚ L'algoritmo del programma

Dopo i comandi relativi ai dati di input, ecco il cuore del programma, vale a dire l'algoritmo che risolve il problema che si vuole affrontare nel codice. In questo caso si ha semplicemente il prodotto dei due numeri `c= a*b`. Alla variabile `c` viene assegnato il risultato del prodotto dei valori delle variabili `a` e `b`.

✚ Stampa dei risultati

Occorre ora mostrare il risultato. Abbiamo di nuovo un'istruzione di `write`. Difatti, il comando seguente:

```
write(6,*) a
```

rende disponibile all'esterno il valore numerico assunto dalla variabile `a`.

Perciò noi scriviamo:

```
write(6,*) 'Il prodotto di ', a, ' e ', b, ' e'':  ', c
```

Tra apici (apostrofi) scriviamo le stringhe di caratteri che vogliamo visualizzare così come sono. Scriviamo invece al di fuori degli apici le variabili di cui vogliamo visualizzare il valore numerico.

Osserviamo che per scrivere la e con l'accento dobbiamo porre due apici uno di seguito all'altro.

5 Struttura di un programma

Abbiamo visto che un codice in Fortran inizia con la dichiarazione di `program`, seguito dal nome del codice, e termina con la dichiarazione di `end`. Queste due istruzioni racchiudono il vero e proprio programma.

Tenendo conto del programma di esempio che abbiamo visto, una tipica struttura di programma è la seguente

```
program NOMEPROGRAMMA
implicit none
dichiarazione delle variabili
definizione dei parametri
assegnazione dei dati di input
istruzioni per l'algoritmo di risoluzione del problema
stampa dei risultati
end
```

I punti relativi al comando `implicit none` e alla dichiarazione delle variabili sono stati già ampiamenti discussi nella Sezione 4.

Analizziamo quindi gli altri punti.

6 Definizione dei parametri

Nell'analisi di un problema che deve essere risolto numericamente al calcolatore attraverso l'esecuzione di un opportuno codice scritto in Fortran, può accadere di avere a che fare con delle costanti, cioè con delle variabili che non possono essere mai cambiate durante l'esecuzione del programma stesso. Ad esempio costanti come π o e, che non sono definite intrinsicamente nel Fortran, devono essere definite, se servono, all'interno del programma.

A tale scopo, **prima** si deve dichiarare il tipo di parametro (se `integer`, `real`,...) e subito **dopo** gli si deve assegnare il valore numerico. Queste due operazioni devono essere fatte nella parte iniziale del programma, dove vengono dichiarate le variabili.

Esempio

```
real pi, e
parameter( pi=3.1415927, e=2.7182818)
```

7 Principali strutture in Fortan

7.1 Struttura sequenziale

Si ha una struttura sequenziale quando le istruzioni devono essere eseguite tutte in successione, senza saltarne una.

Esempio

```
c=a*b
d=a+b
e=b-a
```

Programmare utilizzando solamente una struttura sequenziale, tuttavia, non permette la risoluzione di problemi anche molto semplici come trovare il massimo/minimo tra due elementi, o calcolare la somma di un certo numero di termini.

Si hanno, perciò, le strutture alternative e i cicli.

7.2 Struttura alternativa

La struttura alternativa permette di eseguire un'istruzione piuttosto che un'altra a seconda che sia vera o falsa una determinata condizione.

Esempio

Si vuole calcolare il massimo tra due elementi `a` e `b` memorizzando nella variabile `max` il massimo. L'algoritmo da tradurre in linguaggio Fortran è il seguente:

- se $a \geq b$ allora
  - $max = a$
- altrimenti
  - $max = b$

Abbiamo un'espressione logica, data da $a \geq b$: se l'espressione logica è vera allora si esegue un comando, altrimenti si esegue un'altra istruzione.

In Fortran scriviamo

```
if (a.ge.b) then
     max=a
else
     max=b
end if
```

Diversi tipi di strutture `if` sono:

- se è vera un'espressione logica allora si eseguono i comandi `comando1, comando2, ... comandoN` altrimenti non si esegue nulla:

```
if (espressione logica) then
     comando1
     comando2
     ....
     comandoN
end if
```

- se, vera un'espressione logica, si deve eseguire **un solo comando eseguibile** e, se falsa, non si deve fare nulla, si può scrivere (è l'*if logico*):

```
if (espressione logica) (comando eseguibile)
```

- se è vera un'espressione logica allora si eseguono determinati comandi altrimenti, se è vera un'altra espressione logica, si eseguono altri tipi di comandi, altrimenti si eseguono altri comandi... Questa forma più generale si traduce in:

```
if (espressione logica 1) then
     comandi
elseif (espressione logica 2) then
     comandi
     :
     :
else
     comandi
endif
```

Gli *operatori di condizione* che possono essere utilizzati nelle espressioni logiche sono dati in tabella:

operatore	esempio	descrizione
`.gt.`	`x.gt.y`	vera se $x > y$
`.lt.`	`x.lt.y`	vera se $x < y$
`.eq.`	`x.eq.y`	vera se $x = y$
`.ne.`	`x.ne.y`	vera se $x \neq y$
`.le.`	`x.le.y`	vera se $x \leq y$
`.ge.`	`x.ge.y`	vera se $x \geq y$

Un'espressione logica può essere formata da più espressioni logiche messe insieme da un *operatore logico*:

operatore	esempio	descrizione
`.not.`	`(.not.x.eq.0)`	vera se $x \neq 0$
`.and.`	`(x.ge.0.and.x.le.1)`	vera se $0 \leq x \leq 1$
`.or.`	`(x.le.0.or.x.ge.1)`	vera se $x \leq 0$ o $x \geq 1$

Esempio

Si vuole scrivere un programma che, dato un numero x, assegna alla variabile `ret` il valore $+1$ se $0 < x < 1$, il valore -1 se $x \in]-\infty, 0[\cup]1, +\infty[$, e 0 altrimenti.

L'algoritmo che traduce le condizioni ora dette si scrive in Fortran come:

```
if (x.gt.0.and.x.lt.1) then
     ret=1
elseif (x.lt.0.or.x.gt.1) then
     ret=-1
else
     ret=0
endif
```

7.3 Cicli

Quando un certo numero di istruzioni deve essere ripetuto più volte (o un certo numero n di volte o finchè una certa espressione logica sia verificata) allora si ha una struttura ciclica.

7.3.1 Ciclo `do while`

Si voglia tradurre in Fortran un algoritmo espresso mediante la frase: fino a quando è vero questa espressione logica (o predicato) allora devono essere eseguite queste istruzioni. Nel momento in cui il predicato diventa falso allora non si eseguono più quelle istruzioni.

Si ha:

```
do while (predicato)
     istruzioni
end do
```

La prima volta che si va nel ciclo `do while` si valuta il predicato: se è vero si eseguono le istruzioni, se falso si va all'istruzione successiva alla riga di `end do`. Se sono eseguite le istruzioni del ciclo, si valuta nuovamente il predicato e se è vero si eseguono nuovamente le istruzioni del ciclo, altrimenti si esce dal ciclo stesso. Si va avanti in questo modo fino a quando il predicato risulta essere falso.

Il rischio è che il ciclo potrebbe durare all'infinito, e quindi bisogna prestare molta attenzione sul fatto che il predicato non possa essere sempre vero! D'altra parte se il predicato è sempre falso non si entra mai nel ciclo `do while` e quindi quelle istruzioni non sono mai eseguite.

Esempio

Si voglia calcolare un'approssimazione del punto fisso della funzione $g(x) = \frac{1}{\sin x}$ con un'accuratezza dell'ordine di 10^{-6}. Partendo da un valore iniziale x_0 assegnato, si calcola la successione $x_{k+1} = g(x_k)$ e si prosegue fino a quando $|x_{k+1} - x_k| < eps$.

Perciò, fintantochè $|x_{k+1} - x_k| \geq eps$ allora il metodo del punto fisso viene iterato. Se il punto iniziale, tuttavia, non è scelto nell'intervallo di convergenza del metodo le iterazioni potrebbero andare avanti all'infinito. Per evitare questo inconveniente, si fa un controllo anche sul numero massimo di iterazioni ammissibili per arrivare a

convergenza (numero massimo scelto a seconda del problema). Perciò, si va avanti nel ciclo `do while` fintantochè $|x_{k+1} - x_k| \geq eps$ textbfe il numero di iterazioni eseguite è minore del numero massimo. Il predicato diventa falso quando una delle due espressioni risulta falsa, cioè quando $|x_{k+1} - x_k| < eps$ (cioè siamo arrivati ad un'approssimazione del punto fisso) oppure il numero di iterazioni uguaglia il numero massimo (e in tal caso bisogna vedere se il metodo stia convergendo molto lentamente o se stia divergendo).

In Fortran, l'algoritmo può essere scritto nel programma seguente:

```
      program puntofisso
C
C     programma per la ricerca del punto fisso della funzione
C     1/sin(x)
C     accuratezza richiesta eps=1.e-06
C     numero massimo di iterazioni itmax=50
C     it indica quante volte si entra nel ciclo do while
C
      implicit none
      integer it,itmax
      real xold, xnew, eps,err
      parameter (itmax=50, eps=1.e-06)
      it=1
      write(6,*) 'x iniziale'
      read(5,*) xold         !  corrisponde a x_0
      xnew=1/sin(xold)       !  corrisponde a x_1
      err=abs(xnew-xold)
      do while (err.ge.eps. and. it.le.itmax)
C
C conservo in memoria solo gli ultimi due valori della successione di
C valori del metodo del punto fisso
C xold corrisponde a x_k
C xnew corrisponde a x_{k+1}
          xold=xnew
          xnew=1/sin(xold)
          err=abs(xnew-xold)
          it=it+1  !iterata corrente: k+1
      end do

      write(6,*) 'approssimazione ', xnew, 'in iterate', it
      write(6,*) 'accuratezza ', err
      end
```

7.3.2 Istruzione goto

Il ciclo do while può essere scritto in modo alternativo utilizzando l'istruzione goto che permette di andare dalla riga corrente, in cui si trova l'istruzione goto, ad una riga diversa contrassegnata da una specifica *label* indicata dopo il comando goto e richiamata nelle colonne da 2 a 5 della riga in cui si deve saltare.

Il goto si definisce anche *salto incondizionato*.

Esempio

```
      goto 11
  10 write(6,*) ' Ricomincio il codice '
      :
      :
! qui ci sono linee di comando del codice
      :
      :
  11 write(6,*) 'Scrivi 1 se vuoi tornare indietro: '
      read(5,*) i
      if (i.eq.1) goto 10
      end
```

Il primo goto è quello proprio di salto incondizionato, che porta direttamente da dove ci si trova alla fine del codice, nella riga contrassegnata con la *label* 11. Il secondo goto è sempre di salto incondizionato ma inserito in un *if logico* e porta alla riga contrassegnata dalla *label* 10.

Il ciclo do while si può scrivere in maniera alternativa combinando if e goto nel modo seguente:

```
  10  if (predicato) then
         istruzioni
         goto 10
      endif
```

7.3.3 Ciclo do

Vi sono situazioni in cui una o più istruzioni devono essere ripetute un certo numero di volte e con un certo passo.

Si ha il ciclo

```
do indice=inizio, fine, step
    istruzioni
end do
```

dove `indice` è una variabile che indica il contatore del ciclo `do`, e `inizio`, `fine` e `step` sono variabili o espressioni numeriche che indicano, rispettivamente, il valore iniziale e il valore finale che devono essere assunti dalla variabile contatore `indice` e l'incremento da dare a `indice` ad ogni iterazione.

L'incremento sulla variabile `indice` è fatto dopo che sono state eseguite le istruzioni del ciclo stesso. Se l'incremento è uguale a 1, si può evitare di scriverlo perchè è messo di *default*.

Quando si usa un ciclo `do` non è consentito scrivere istruzioni `goto` che rimandano dall'esterno all'interno del ciclo. E non è consigliato rimandare dall'interno all'esterno del ciclo con istruzioni `goto`.

Esempio

Si vuole calcolare la somma dei primi 100 numeri pari: $2 + 4 + 6 + ... + 100$.

L'algoritmo è il seguente:

```
  integer i,m
  :
  :
! righe di istruzioni del codice
  :
  :
  m = 0 ! variabile che contiene la somma dei primi 100 numeri
  do i=2,100,2
    m= m+i
  end do
```

Osserviamo come sia nelle strutture alternative, sia nei cicli, abbiamo scritto le istruzioni *indentellate* rispetto all'apertura e alla chiusura della struttura stessa: le istruzioni sono rientrate rispetto a `if-else-end if` o `do while-end do` o `do -end do`: in questo modo è ben visibile graficamente il tipo di struttura con le relative istruzioni.

Perchè questa attenzione di *stile*?

Nel caso in cui il codice sia stato scritto con errori di sintassi, è più facile trovare gli errori (per esempio è più facile individuare un ciclo `do` che non è stato chiuso).

Inoltre il codice diventa più leggibile sia per chi lo ha scritto sia per chi lo legga la prima volta.

8 Operazioni di input e output

Quando si hanno molti dati di input e output, come, ad esempio, matrici e vettori, conviene scrivere i dati di input in un file, aprire questo file all'interno del programma e leggere i dati scritti nel file. Analogamente, al posto di visualizzare i dati di output sul terminale, li si scrive su un file che verrà successivamente aperto con un *editor* per essere visualizzato.

Le operazioni fondamentali che ci servono, senza entrare nei dettagli, sono le seguenti:

- scrivere il file di dati, e salvarlo con un nome opportuno, per esempio `dati.dat`
- nel codice sorgente, dopo la dichiarazione delle variabili scrivere:

  ```
  open(unit1, FILE='dati.dat', STATUS='status')
  ```

  dove `unit1` è un numero intero che può variare nell'intervallo $[1, 999]$ che sarà associato al file `dati.dat` all'interno del programma, e `status` indica lo stato del file, se esiste già (`old`), se non esiste ancora (`new`) o se non si conosce il suo stato (`unknown`).

  Un comando simile a questo deve essere dato se vogliamo scrivere i risultati del codice su un file di output, ad esempio `risultati.ris`, facendo attenzione ad associare ad esso un etichetta diversa di quella data al primo file (`unit2` diverso da `unit1`):

  ```
  open(uniti2, FILE='risultati.ris', STATUS='status')
  ```

- la lettura dei dati di input è quindi fatta tramite il comando

  `read(unit1, format)` *lista delle variabili del file dati.dat*

  `format` indica il tipo di formato di lettura (o scrittura) dei dati. Generalmente in lettura si lascia il formato standard, ponendo un `*`.

- la scrittura dei dati di output è fatta invece come

  `write(unit2, format)` *lista dei risultati da scrivere in risultati.ris*

- i files che sono stati aperti vengono chiusi, prima della `end` finale del programma con il comando

  ```
  close(unit1)
  close(unit2)
  ```

Esempio

Nel file `dati.dat` si scrivano i dati relativi ai valori da assegnare alle variabili `a, b, c, i, n`:

```
12.5    15.8
81.89
2       100
```

Nel codice scriveremo:

```
open(10, FILE='dati.dat', STATUS='old')
open(11, FILE='risultati.ris', STATUS='unknown')
```

```
      read(10,*) a, b
      read(10,*) c
      read(10,*) i, n
      :
      :
      :
C     algoritmo del programma
C
      :
      :
C     stampa dei risultati
C
      write(11,*) 'variabile f ', f
      write(11,*) 'variabile g ', g
      write(11,*) 'variabile j ', j

      close(10)
      close(11)

      end
```

Una volta fatto girare il codice i risultati saranno scritti nel file `risultati.ris`. Nell'apertura di questo file abbiamo scritto `status='unknown'` anzichè `new` perchè, se facciamo girare il codice più di una volta, non siamo costretti a cancellare il file di risultati prima di lanciare il codice in modo da rendere il suo stato `new`, ma lo sovrascriviamo.

Occorre prestare molta attenzione perchè ci sia concordanza tra il significato delle quantità numeriche scritte nel file di input e le variabili lette dal codice.

Senza entrare nei dettagli dei meccanismi di lettura di un file, ci basta sapere che ogni riga di comando `read` determina la lettura di una riga del file di input, e quindi, dobbiamo far acquisire le variabili contenute riga per riga con tanti comandi di `read` quante sono le righe che nel file di dati contengono variabili, proprio come è stato fatto nell'esempio.

Per essere sicuri di aver letto bene i dati, li si può scrivere nel file di output, in modo da avere in questo unico file sia l'input che l'output del problema che si risolve.

9 Vettori e matrici

Il Calcolo Numerico affronta spesso problemi definiti non su variabili *scalari* ma su variabili *vettoriali* o *matriciali*.

Si consideri il problema della soluzione di sistemi lineari, per rendersi conto

di quanto sia importante poter lavorare con strutture di dati che ci permettono di definire vettori e matrici.

Ma basti anche pensare al semplice problema di fare la somma di n numeri reali: se questi numeri sono memorizzati in un vettore, fare la loro somma sarà un'operazione semplice altrimenti sarà abbastanza complicato, specie se n è un numero elevato: se essi sono memorizzati come un vettore di dimensione n, che chiamiamo v, per fare la somma si ha l'algoritmo:

```
somma=0.0
do i=1,n
   somma=somma+v(i)
end do
```

Se una matrice ha dimensione $n \times m$, anzichè memorizzare i suoi elementi come $n \times m$ variabili reali (facendo bene attenzione alla disposizione di riga e colonna di ciascuna variabile nella matrice stessa - si veda l'esempio del calcolo del determinante di una matrice vista a pagina 3), considereremo la matrice `A` di dimensione $n \times m$ e di elementi reali `A(i,j)`, `i=1,...,n`, `j=1,...,m`.

Poichè in Fortran 77 la allocazione delle variabili è statica, nella dichiarazione di queste variabili occorre dare una dimensione massima.

Esempio

```
      program vettore
      implicit none
      integer i
      integer n     !dimensione effettiva del vettore v
      real v(10)    !10 e' la dimensione massima del vettore v

      open(10, FILE='dati.dat', STATUS='old')
      open(11, FILE='risultati.ris', STATUS='unknown')

C   lettura del vettore dal file dati.dat
      read(10,*) n
      do i=1,n
        read(10,*) v(i)
      end do
      write(11,*) 'vettore v:'
      write(11,*) 'dimensione :', n
      do i=1,n
        write(11,*) 'componente', i, '= ', v(i)
      end do

      close(10)
      close(11)
```

```
      end
```

Le componenti del vettore sono lette tramite un ciclo `do-end do`, quindi riga per riga. Il file `dati.dat` deve essere scritto riga per riga: la prima riga deve contenere il valore della dimensione del vettore e le righe successive le componenti del vettore stesso.

```
5
1.
2.
3.
4.
5.
```

Allora il file `risultati.ris` conterrà l'output:

```
 vettore v:
 dimensione : 5
 componente 1=   1.
 componente 2=   2.
 componente 3=   3.
 componente 4=   4.
 componente 5=   5.
```

La dimensione massima del vettore può essere data anche attraverso un parametro:

```
      integer nmax
      parameter(nmax=10)
      integer n
      real v(nmax)
```

L'uso del parametro è utile quando si hanno più vettori e matrici con la stessa dimensione massima: se si vuole cambiare la dimensione massima si cambia solo il valore del parametro.

Vediamo ora come memorizzare anche le matrici, facendo uso di un comando di lettura più compatto.

Esempio

Sia dato il file `dati.dat`

```
2  3
1. 2. 3.
4. 5. 6.
```

La dichiarazione e la lettura/scrittura di questi dati viene fatta nel modo seguente:

```
C dichiarazione delle variabili
      integer nmax
      parameter(nmax=10)
      integer n,m
      real A(nmax,nmax)

C lettura delle variabili

      open(10, FILE='dati.dat', STATUS='old')
      open(11, FILE='risultati.ris', STATUS='unknown')
      read(10,*) n,m
      do i=1,n
            read(10,*) ( A(i,j), j=1,m)
      end do
C scrittura delle variabili
      do i=1,n
            write(11,*) ( A(i,j), j=1,m)
      end do
```

La matrice `A` è stata scritta nel file di dati come siamo abituati a vederla, disposta su `n` righe ed `m` colonne. Il comando `read(10,*) ( A(i,j), j=1,m)` è una sorte di `do` implicito che permette di leggere tutti gli elementi della riga `i`-sima.

Esempio

Vediamo ora un programma d'esempio sull'uso delle matrici e dei vettori: scriviamo un codice che esegua il prodotto di una matrice per un vettore. Data una matrice A $n \times m$ e un vettore x di dimensione m, il vettore prodotto $y = Ax$ ha n componenti date da:

$$y_i = \sum_{j=1}^{m} A_{ij} x_j, \quad i = 1, \ldots, n$$

I dati di input sono:

- la dimensione della matrice: `n` righe e `m` colonne
- i valori delle componenti della matrice
- i valori delle componenti del vettore (di dimensione `m`).

I dati di output:

- i valori delle componenti del vettore prodotto (di dimensione `n`).

Il codice è il seguente:

```
      program matvett
C
C   programma per il calcolo del prodotto matrice vettore
C

      implicit none
      integer nmax,mmax
      parameter (nmax=20, mmax=20)
      integer i,j, n,m
      real A(nmax,mmax), xvet(mmax), yvet(nmax)

      open(8,file='datimatvet.dat', status='old')
      open(9,file='prodmatvet.ris', status='unknown')

      read(8,*) n,m
      write(9,*) 'Prodotto matrice-vettore'
      write(9,*) 'dimensioni della matrice ', n, m
      write(9,*) 'matrice A:'

      do i=1,n
         read(8,*) (A(i,j),j=1,m)
         write(9,*) ( A(i,j),j=1,m)
      end do
       write(9,*) 'vettore x:'
      do i=1,m
         read(8,*)  xvet(i)
         write(9,*) xvet(i)
      end do

       write(9,*) 'vettore prodotto y:'
      do i=1,n
C  inizializziamo a 0. ciascuna componente del vettore prodotto
            yvet(i)=0.0
C  eseguiamo un ciclo do per j=1,m per calcolare le componenti
C  del vettore prodotto
            do j=1,m
                yvet(i) = yvet(i) + A(i,j)*xvet(j)
            end do
C  scriviamo ciascuna componente sul file di risultati
        write(9,*)  yvet(i)
      end do

     close(8)
```

```
        close(9)

        end
```

Al file di dati

```
2  3
1.0  0.0  0.0
2.0  1.0 -1.0
1.
1.
1.
```

corrisponde il file di risultati

```
 Prodotto matrice-vettore
 dimensioni della matrice  2 3
 matrice A:
  1.  0.  0.
  2.  1. -1.
 vettore x:
  1.
  1.
  1.
 vettore prodotto y:
  1.
  2.
```

10 Funzioni intrinseche

Nello scrivere programmi in Fortran, molto spesso devono essere utilizzate delle funzioni matematiche di base quali $\sin$, $\cos$, $\sqrt{\cdot}$ ...

In Fortran sono presenti delle funzioni intrinseche che è possibile chiamare direttamente all'interno del codice (si veda l'esempio del programma della ricerca del punto fisso, dove abbiamo utilizzata le funzioni $\sin x$ e $|x|$).

Diamo, nel seguito, un elenco di alcune funzioni che possono essere utili nei programmi e che sono intrinsiche del Fortran:

funzione	descrizione
`SQRT(x)`	calcola la radice quadrata di `x`
`LOG(x)`	calcola il logaritmo naturale di `x`
`LOG10(x)`	calcola il logaritmo in base 10 di `x`
`EXP(x)`	calcola la funzione esponenziale e^x
`SIN(x)`	calcola $\sin x$ (espresso in radianti)
`COS(x)`	calcola $\cos x$ (espresso in radianti)
`TAN(x)`	calcola $\tan x$ (espresso in radianti)
`ASIN(x)`	calcola $\arcsin x$ (espresso in radianti)
`ACOS(x)`	calcola $\arccos x$ (espresso in radianti)
`FLOAT(x)`	converte l'argomento in un valore reale
`INT(x)`	converte l'argomento in un valore intero
`ABS(x)`	calcola il valore assoluto di x
`MAX(x1,x2,...,xN)`	calcola il massimo valore di `x1,x2,...,xN`

11 Introduzione ai sottoprogrammi

Molti problemi richiedono l'uso di funzioni che sono date dalla combinazione di funzioni intrinseche del Fortran, che devono essere richiamate più volte all'interno del codice stesso e che possono essere più o meno complicate.

Un esempio di funzione complicata è dato dalla seguente

$$f(x,y) = \tan\delta - \tan x/\sqrt{1+\tan^2\Gamma\tan^2 x}$$

dove $\sin\Gamma = \sin y\cos\gamma - \sin\gamma\sqrt{\cos^2 y + \tan^2 x}$ e δ e γ sono dei parametri assegnati.

Come fare a scrivere in Fortran funzioni come queste?

E, soprattutto, se dobbiamo richiamare queste funzioni su più righe del codice, come fare a non dover riscrivere ogni volta tutta la formula che definisce la funzione ma, semplicemente, richiamarla, in qualche modo?

Possiamo ora intuire l'importanza dei sottoprogrammi: dividere un problema complicato in sottoproblemi più piccoli. La risoluzione dei problemi più piccoli viene organizzata in modo da risolvere il problema originario.

Se riusciamo, infatti, a scrivere una funzione, in maniera separata rispetto al programma principale, che traduca, in Fortran, la $f(x,y)$ scritta prima, e se riusciamo a richiamarla nel programma principale, là dove serve, con una semplice istruzione, del tipo `a=f(x,y)`, allora il problema sarà semplificato nella sua impostazione e nella sua risoluzione.

In Fortran esistono due tipi di sottoprogrammi: le `functions` e le `subroutines`.

11.1 Struttura di una `function`

Una `function` è un programma separato e completo rispetto al programma principale, ma con una struttura di inizio e fine diversa rispetto al programma princi-

pale. Essa calcola un singolo valore che viene passato nel programma principale e assegnato alla variabile che chiama la `function` stessa (vediamo adesso come).

Una `function` può assumere valori di un certo tipo (per esempio reale o intero): bisogna tenere conto di ciò nello scrivere una `function`.

Lo schema è infatti il seguente:

```
tipo function NOMEFUNCTION( lista delle variabili formali)
implicit none
dichiarazione delle variabili
istruzioni
NOMEFUNCTION = assegnazione da dare alla function
return
end
```

La `function NOMEFUNCTION` dipende da una o più variabili che vengono introdotte tra parentesi nella prima riga. Il corpo della `function` non differisce molto da quello del programma principale. L'importante è che ci sia almeno una istruzione che assegna alla `function` il suo valore: `NOMEFUNCTION = ...` Il comando `return` fa sì che il valore assegnato a `NOMEFUNCTION` venga passato nel programma principale, là dove è stata chiamata la `function`.

Vediamo un esempio per chiarire il concetto.

Esempio

Nel programma del punto fisso che abbiamo già scritto, vogliamo scrivere la funzione $1/\sin x$ come una `function`. Possiamo scrivere la `function` nello stesso file che contiene il programma principale, subito dopo la `end` del programma.

```
real function g(x)
implicit none
real x

g= 1/sin(x)

return
end
```

Il programma principale, invece, va modificato in questo modo

```
program puntofisso
implicit none
integer it,itmax
real xold, xnew, eps,err, g
parameter (itmax=50, eps=1.e-06)
it=1
```

```
write(6,*) 'x iniziale'
read(5,*) xold          !  corrisponde a x_0
xnew=g(xold)            !  corrisponde a x_1
err=abs(xnew-xold)
do while (err.ge.eps. and. it.le.itmax)
    xold=xnew
    xnew=g(xold)
    err=abs(xnew-xold)
    it=it+1  !iterata corrente: k+1
end do
 :
 :  tutto il resto invariato
end
```

Nel programma principale abbiamo definito la `function` come variabile reale. E per chiamarla scriviamo semplicemente `xnew= g(xold)`.

Osserviamo che la variabile che diventa l'argomento della `function` nel programma principale può non avere lo stesso nome formale nel sottoprogramma `function` ma ha lo stesso significato: nell'esempio visto abbiamo `xold` nel programma principale e `x` nella `function`, due nomi diversi ma che rappresentano la stessa variabile.

11.2 Struttura di una `subroutine`

La `subroutine` è un tipo di sottoprogramma simile alla `function` anche se differisce da essa per molti aspetti. Innanzitutto non c'è nessun tipo da dichiarare all'inizio come per la `function` e non c'è nessun valore associato al nome della `subroutine` ma accanto alla lista dei parametri che sono utilizzati all'interno della `subroutine` come dati di input si pone anche la lista dei parametri di output. Nel programma principale, una subroutine è chiamata attraverso il comando `call`.

```
subroutine NOMESUBROUTINE( lista delle variabili )
implicit none
dichiarazione delle variabili
istruzioni
return
end
```

Esempio

Nel programma che calcola il prodotto di una matrice per un vettore scriviamo in una `subroutine` la parte relativa al prodotto stesso:

```
subroutine prodotto(nmax,n,m,A,x,y)
implicit none
integer i,j,n,m,nmax
real A(nmax,m), x(m), y(n)
```

```
      do i=1,n
          y(i)=0.0
          do j=1,m
             y(i) = y(i) + A(i,j)*x(j)
          end do
      end do
      return
      end
```

Osserviamo che nella lista dei parametri della `subroutine` dobbiamo inserire tutte le variabili che servono per ottenere il vettore prodotto: non solo la matrice `A` e il vettore `x` ma anche le dimensioni. Per i vettori possiamo dare la dimensione effettiva e scrivere `x(m)` e non `x(nmax)`. Per la matrice, poichè la memorizzazione di una matrice avviene per colonne, è necessario lasciare la dimensione massima sulle righe, quindi dichiare `A(nmax,m)`, o `A(nmax,nmax)`, per evitare che i dati vengano passati in maniera scorretta.

Nel programma principale dobbiamo fare queste modifiche

```
      program matvett
      :
      :
C tutte le istruzioni come prima
      :
      write(9,*) 'vettore prodotto y:'
      call prodotto(nmax,n,m,A,xvet,yvet)
      do i=1,n
         write(9,*)  yvet(i)
      end do
      end
```

Nel programma principale, quindi, chiamiamo la `subroutine` mediante il comando `call prodotto(nmax,n,m,A,xvet,yvet)`.

I nomi delle variabili non sono identici nel programma principale e nella `subroutine` ma hanno lo stesso significato!

12 Cenni sull'istruzione `format`

L'istruzione `format` è utile soprattutto per produrre un output elegante a vedersi. Se, l'istruzione di `write(unit,*)` è sostituita da `write(unit, label)` allora, una linea del codice deve essere contrassegnata dalla etichetta `label` (un numero) nelle colonne da 2 a 5, seguita dall'istruzione che definisce il formato di scrittura.

```
         write(unit,label) {variabili o stringa da stampare}
 label   format( {istruzioni sul formato} )
```

Esempio

```
      do i=1,n
      write(11,100) i, yvet(i)
      end do
100   format(1x, i3, 1x, e13.6)
```

Nell'istruzione format indichiamo di lasciare 1 spazio bianco (1x), di scrivere una variabile intera usando 3 caratteri (i3), di lasciare uno spazio bianco (1x) e di scrivere una variabile intera in formato esponenziale, usando 13 caratteri, di cui 6 per la mantissa (e13.6).

Nel file di risultati se il vettore yvet ha componenti 1. e 2. leggeremo:

```
1  0.100000E+01
2  0.200000E+01
```

Vediamo i vari tipi di formati:

formato	descrizione
iN	stampa di numeri interi usando N caratteri
eM.N	stampa di numeri reali in formato esponenziale usando M caratteri, di cui N per la mantissa: precisione semplice
fM.N	stampa di numeri reali in formato fisso M caratteri sono in virgola fissa N caratteri sono per le cifre decimali
aN	formato alfanumerico per stringhe di N caratteri
Nx	stampa N caratteri bianchi (vuoti)
/	serve per andare a capo nella stampa

Se la variabile richiede più caratteri di quelli dati per essere stampata, allora la stampa non risulta corretta ma contrassegnata da asterischi o da altri caratteri stabiliti dal compilatore.

Se vogliamo stampare più variabili con lo stesso formato basta considerare l'istruzione

format(n{lista dei formati})

dove n è una costante numerica (e non una variabile).

Esempi

```
        write(9,102) 'dimensioni della matrice ', n,m
102     format( a30, /, 20x,2(1x,i2))
```

Il formato contrassegnato dalla *label* 102 riserva 30 caratteri per la stringa dimensioni della matrice, poi va a scrivere nella riga successiva, lasciando 20 spazi bianchi e poi ripetendo per 2 volte il formato di uno spazio bianco e due caratteri per la variabile intera.

Se n = 4 e m= 5, abbiamo l'output

```
     dimensioni della matrice
                     4  5
```

Scriviamo adesso il vettore in forma compatta come abbiamo già fatto per la matrice, e stampiamo 3 componenti alla volta su ogni riga:

```
            write(9,100)  (i, yvet(i), i=1,n)
 100        format(3(1x, i3, 1x, e13.6))
```

Se il vettore yvet ha componenti 1, 2, 6, 1, l'output sarà:

```
 1  0.100000E+02   2  0.200000E+01   3  0.600000E+01
 4  0.100000E+02
```

Un'analoga istruzione per la scrittura di una matrice è:

```
      do i=1,n
          write(9,101) ( A(i,j),j=1,m)
      end do
101   format (3(e13.6))
```

Bibliografia
Testo consigliato per gli approfondimenti:
F. Sartoretto, M. Putti
Introduzione al Fortran
Per applicazioni numeriche
Edizioni Libreria Progetto, Padova