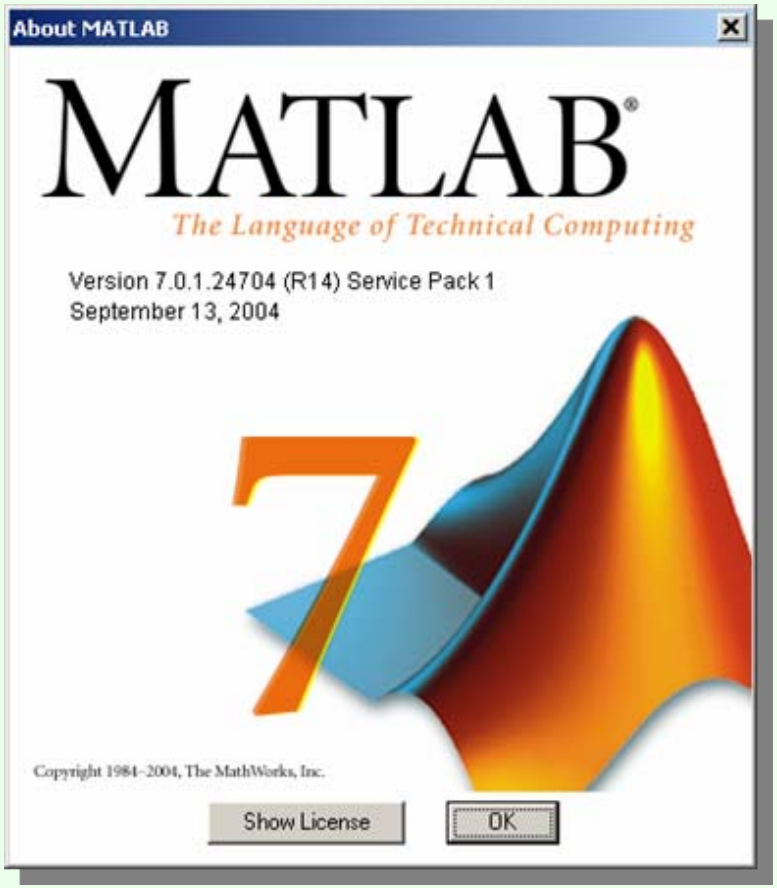
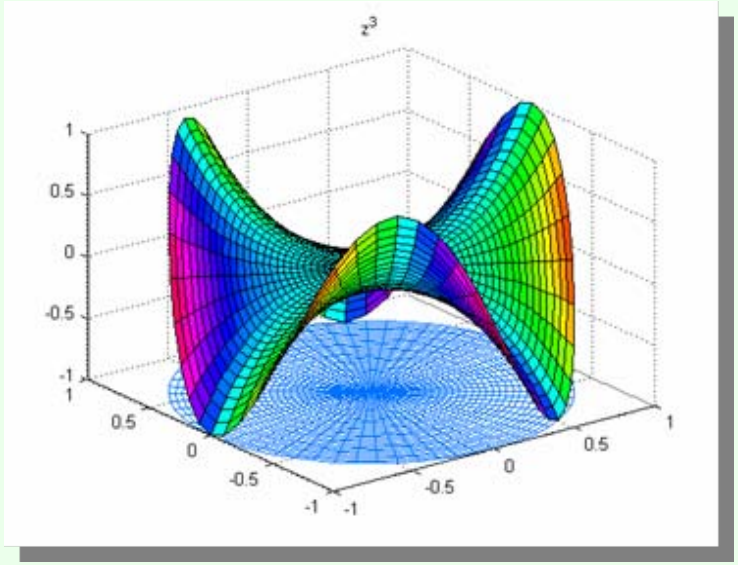
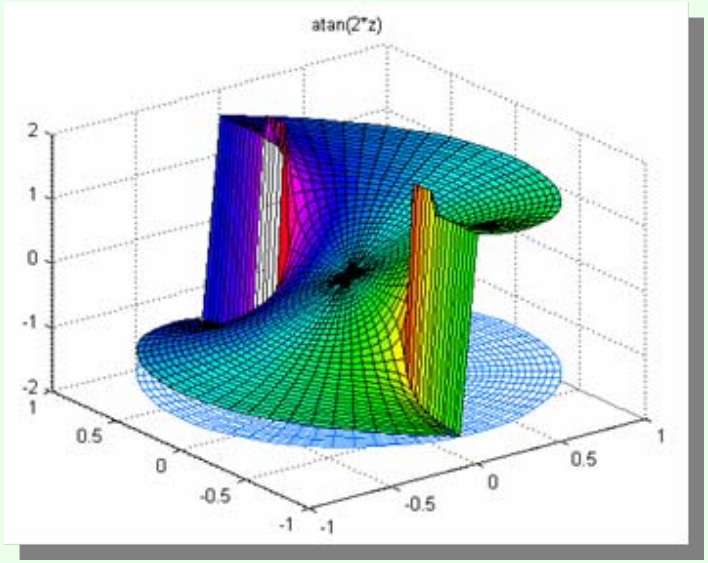
TMA

Tutorial di Matlab Advanced







Puntatori di funzioni

Anche se il titolo potrebbe spaventare l'utente, occorre viceversa sottolineare che l'argomento dei puntatori di funzioni è di grande importanza e più semplice di quanto si possa pensare.

Partiamo dall'esigenza per poi spiegare come implementarla.

Si supponga di avere implementato in linguaggio Matlab™ il metodo dicotomico per la risoluzione di un'equazione algebrica non lineare. Il programma (lo script) chiama all'interno delle iterazioni numeriche la funzione specifica che l'utente ha codificato separatamente in un file avente lo stesso nome.

Qualche tempo dopo nasce l'esigenza di risolvere un nuovo problema basato sull'azzeramento di una funzione algebrica non lineare.

L'utente, programmatore, dovrebbe recuperare il file (script) già testato, modificare ovunque l'occorrenza della chiamata alla prima funzione algebrica per sostituirla con la seconda, quella attuale.

Il difetto di quest'approccio programmativo è che in un unico script coesistono i dati di input relativi allo specifico problema da risolvere e l'algoritmo risolutivo di più generale formulazione ed applicabilità.

Puntatori di funzioni

L'ideale per un utente esterno ma anche per lo stesso programmatore che diviene, in un secondo momento, utente di sé stesso, è quello di poter disporre di una struttura programmativa cosiffatta:

Routine dedicata alla risoluzione del problema numerico specifico (ad esempio azzeramento di funzione tramite metodo dicotomico) in grado di ricevere via lista il nome della funzione da azzerare, l'intervallo di incertezza iniziale $[a,b]$, il numero massimo di iterazioni da effettuare $\mathrm{MAX\_ITER}$, la massima ampiezza finale accettabile dell'intervallo di incertezza δ. Tale routine dovrebbe restituire il punto, c, candidato alla soluzione in accordo con la struttura dell'algoritmo numerico.

L'utente, potendo disporre della routine summenzionata non avrebbe altro che da fornire il nome della funzione da azzerare e gli estremi dell'intervallo di incertezza a, b soddisfacenti la condizione di Bolzano: $f(a)\cdot f(b) \leq 0$ nonché come indicato in precedenza i due parametri $\mathrm{MAX\_ITER}$ e δ.

L'operazione di passare ad una routine il nome di un'altra routine (nel nostro caso una funzione) è definito nel linguaggio programmativo: "**passaggio di un puntatore di funzione**".

Puntatori di funzioni

Per passare via lista ad una routine di Matlab™ il nome di una funzione occorre anteporre al nome della funzione stessa il simbolo: @.

Ad esempio se la funzione si chiama **MyFunc**, il suo puntatore viene passato alla routine **WorkWithFunc** tramite l'istruzione:

```
ret = WorkWithFunc(@MyFunc, ... )
```

Supponiamo che la funzione **MyFunc** sia così strutturata:

```
function f = MyFunc(x)
   f = x ^ 2 * sin(x) – cos(3. * x) + 1.;
```

N.B.: la routine **WorkWithFunc** **non potrà** calcolare, al suo interno, il valore della funzione **MyFunc** in **z = 5**. tramite le istruzioni:

```
z = 5.;
y = MyFunc(z);
```

Puntatori di funzioni

Svolgimento

1. La routine che effettua la somma dei valori di una funzione calcolata in tre punti ha la seguente struttura:

```
function ris = DvdSomma3Valori(FUN,x)
   ris = 0.;
   for i = 1: 3
      y = feval(FUN,x(i));
      ris = ris + y;
   end
```

2. L'utente utilizzerà la routine summenzionata (salvata nel file: **DvdSomma3Valori.m**) tramite il seguente script:

```
clear all
clc
xVect = [-1. 5.  6.];
somma = DvdSomma3Valori(@DvdFunTest,xVect);
disp(['Risultato finale: ',num2str(somma)]);
```

Puntatori di funzioni

Svolgimento continua

3. Si noti che l'utente ha deciso di utilizzare come funzione di lavoro: **`DvdFunTest`** (preventivamente salvata nell'omonimo file: **`DvdFunTest.m`**).

Ecco di seguito un esempio di tale funzione:

```
function f = DvdFunTest(x)
   f = sin(x) * cos(x) – 3. * exp(2. * x – 1.) ^ 2;
```

N.B.: La routine **`DvdSomma3Valori`** potrà essere riutilizzata in qualsiasi altro momento ad esempio con le seguenti chiamate:

```
somma1 = DvdSomma3Valori(@MyFunc,xxx);
somma2 = DvdSomma3Valori(@FunzEse63,xGdl);
pres = DvdSomma3Valori(@FEq,frazMol);
```

È questa l'esemplificazione tipica di riutilizzabilità del codice di calcolo.

Puntatori di funzioni

Ciò che la routine **`WorkWithFunc`** riceve via lista non è esattamente la funzione **`MyFunc`** ma un puntatore a tale funzione.

Per poter effettuare tale calcolo in Matlab™ occorre utilizzare l'istruzione: **`feval`**.

L'istruzione **`yVal = feval(FUNZ, xVal)`** effettua il calcolo della funzione **`FUNZ`** nel punto **`xVal`** e ritorna il risultato in **`yVal`**.

Esempio

Si supponga di dover realizzare una routine in grado di operare su una qualsiasi funzione definita dall'utente. Tale routine dovrà calcolare la somma dei valori della funzione in tre punti assegnati dall'utente fornendo come risultato il valore di tale somma. L'utente dovrà poter specificare alla routine sia il nome della funzione che il vettore di tre elementi contenente i valori della variabile indipendente. La funzione fornita dall'utente dovrà rispettare la struttura:

$$y = f(x)$$

Try – Catch

Nel corso della programmazione si può incorrere nella necessità di tutelarsi da errori inaspettati o imprevedibili. A tale fine si può far ricorso al costrutto:

```
try
   istruzioni;
   istruzioni;
   ...
catch
   istruzioni;
   istruzioni;
   ...
end
```

Più costrutti **`try–catch`** possono essere annidati.

Il programma, quando incontra tale costrutto dapprima inizia ad eseguire la sola porzione di istruzioni contenute nella sezione **`try`**. Se non si verificano degli errori nel corso dell'esecuzione, il programma prosegue con la prima istruzione a valle dell'intero costrutto (cioè dopo **`end`**). Non appena una delle istruzioni contenute nella sezione **`try`** produce un errore, Matlab™ abbandona tale sezione e passa direttamente alla sezione **`catch`**.

Try – Catch

Se un costrutto **`try-catch`** è stato attivato in una routine che in seguito chiama un'altra routine e se all'interno di quest'ultima accade un errore di esecuzione, Matlab™ abbandona direttamente questa routine e torna nella sezione **`catch`** di quella originaria.

Il salto alla prima istruzione **`catch`** gerarchicamente dominante come livello di chiamata avviene anche se le due routine non sono adiacenti nella serie di chiamate. Per maggior chiarezza:

- **`RoutineA`**: contiene un costrutto **`try-catch`** e chiama la **`RoutineB`**
- **`RoutineB`** chiama la **`RoutineC`** che chiama la **`RoutineD`**
- **`RoutineD`** contiene un'istruzione che manda in errore Matlab™
- Matlab™ abbandona la lista di chiamata contenente le routine intermedie e torna direttamente alla **`RoutineA`** eseguendo la sezione **`catch`** presente nella **`RoutineA`**.

N.B.: da una parte il costrutto **`try-catch`** risulta essere molto potente per rendere più robusto e potenzialmente controllabile il codice scritto. Anche gli utenti finali possono così comunicare al programmatore il messaggio di errore gestito dalla sezione **`catch`**. La sezione **`catch`** non solo può emettere messaggi ma anche risolvere la situazione di errore gestendo il caso speciale (soprattutto se il problema numerico ha una valenza fisica).

Try – Catch

N.B.: al contempo il costrutto **`try-catch`** può sviare l'utente ed il programmatore. Si supponga infatti di avere un codice descrivibile tramite la struttura di routine annidate appena delineata.

Se il programmatore all'interno della **`RoutineD`** blocca l'esecuzione con un'istruzione del tipo:

```
error('RoutineD. Portata entrante negativa. Impossibile
            continuare…')
```

Matlab™ non emetterà a video il messaggio di errore della **`RoutineD`**, relativo alla portata negativa, bensì abbandonerà l'intera lista di chiamata e tornando direttamente alla **`RoutineA`** eseguirà l'insieme di istruzioni relative alla sezione **`catch`**.

L'utente ed il programmatore stesso, con grande probabilità resterà spiazzato dalla mancata emissione del messaggio di errore della **`RoutineD`** ed ignorandola si concentrerà sulla ricerca di un errore inesistente nella **`RoutineA`**.

Per ovviare a questo pericolo se si sceglie di operare con una struttura **`try-catch`** è opportuno implementarla all'interno di **tutte** le routine con compongono il programma complessivo. In questo caso diviene dominante il costrutto **`try-catch`** più profondamente annidato nella lista di chiamata.

Variabili globali

Se è necessario utilizzare le stesse variabili o costanti in due porzioni differenti di codice Matlab™ è sufficiente sfruttare il costrutto **`global`**. L'istruzione **`global`** posta in due o più unità differenti di programma (ad esempio in due function anche non appartenenti allo stesso file) permette di condividere le variabili o costanti ed utilizzarle a fini calcolistici. Se una variabile appartenente al costrutto **`global`** viene modificata in un'unita di programma anche tutte le altre unità di codice, che condividono lo stesso costrutto **`global`**, vedranno modificata tale variabile. Esempio:

File prova.m

```
clear all
global ANTA ANTB ANTC
ANTA=16.42; ANTB=2345.; ANTC=-48.84
t = 353.15;
tensioneVap = Pv(t);
.......
```

File pv.m

```
function y = Pv(temp)
   global ANTA ANTB ANTC
   y = exp(ANTA – ANTB / (temp + ANTC))
```

N.B.: le variabili/costanti nell'istruzione **`global`** sono separate da spazi **non** da virgole!

Variabili statiche

Contrariamente ad altri linguaggi (tipicamente Fortran 77 e 90), in Matlab quando si esce da una funzione, le variabili locali (interne cioè alla funzione stessa) perdono il loro valore (come avviene in C e C++). Ciò significa che quando si chiama nuovamente la funzione, tali variabili non hanno il valore che avevano in precedenza. Se si desidera invece mantenere il valore di tale variabile tra una chiamata e quella successiva è sufficiente definirla "**`persistent`**". Così facendo le variabili "**`persistent`**" manterranno in memoria il loro valore nel corso delle varie chiamate di funzione. In tal senso c'è un'affinità di allocazione di memoria con l'istruzione "**`global`**" vista in precedenza ma contrariamente ad essa le variabili "**`persistent`**" hanno visibilità soltanto interna alla funzione. Esempio:

```
function y = MyFun(x)

   persistent iConto jConto xyz

   iConto = iConto + 1; jConto = jConto + 5;

   xyz = xyz + sin(x + jConto)

   y = cos(xyz)
```

N.B.: come nel caso di **`global`** le variabili sono separate da spazi e **non** da virgole!

Struttura dei file

Spesso accade che si debba realizzare un file principale di script in cui effettuare un calcolo che debba utilizzare una specifica funzione scritta dall'utente. Ciò è quanto rappresentato schematicamente nell'esempio precedente (TMA-11).

I vari libri e manuali su Matlab™ indicano la strada di realizzazione di due file distinti. Il primo contenente lo script ed il secondo contenente la funzione ed avente lo stesso nome.

In realtà l'utente preferirebbe poter raccogliere in uno stesso file sia lo script che la funzione utilizzata dallo script stesso.

In poche parole si desidera poter scrivere in uno stesso file una sezione principale (**`PROGRAM`** in Fortran 77 e 90, **`main`** in C e C++) ed una o più funzioni ausiliarie.

Ciò è possibile iniziando il file con l'equivalente del **`PROGRAM`** o **`main`** tramite una funzione senza ritorno il cui nome è libero (si consiglia di dare a tale funzione lo stesso nome del file). Alla fine dell'unità principale (la funzione senza ritorno) si pone un'istruzione "**end**" e quindi si prosegue con la definizione della vera e propria funzione ausiliaria definita dall'utente. Ogni funzione deve terminare con un'istruzione "**end**".

Nella pagina seguente è riportato un semplice esempio tratto dalla pagina TMA-11.

Struttura dei file

File termo.m

```
function Termo
   clear all
   global ANTA ANTB ANTC
   ANTA=16.42; ANTB=2345.; ANTC=-48.84
   t = 353.15;
   tensioneVap = Pv(t);
   .......
end

function y = Pv(temp)
   global ANTA ANTB ANTC
   y = exp(ANTA – ANTB / (temp + ANTC))
end
```

N.B.: è anche possibile eliminare le istruzioni “end” alla fine di ogni unità di programma (function). Se si toglie l’istruzione “**end**” ad una unità è necessario farlo per tutte. In altre parole occorre mantenere lo stesso stile programmativo lungo tutto il file.

N.B.: nel file possono essere introdotte più funzioni. La visibilità delle funzioni che seguono la prima è comunque relativa al solo ambito del file stesso. Le funzioni dalla seconda in poi non sono quindi visibili e tantomeno utilizzabili fuori dal file stesso in cui sono state codificate.

Creazione di un file eseguibile

È possibile creare un file eseguibile *standalone* a partire da un file sorgente scritto in Matlab™. Con il termine *standalone* si intende un file (.exe) che può essere eseguito su un qualsiasi computer senza la necessità che su quest'ultimo sia installato Matlab™.

La creazione di un file eseguibile permette, inoltre, di mantenere occultato il contenuto del file sorgente (.m) qualora non si intenda far conoscere l'effettivo contenuto modellistico e teorico del progetto realizzato.

Per generare un file eseguibile a partire da un sorgente Matlab™ è sufficiente utilizzare il compilatore C/C++ di Matlab™ che trasforma il sorgente .m in un file .c e quindi lo compila e ne fa il link utilizzando un compilatore interno o uno già disponibile (ad esempio Microsoft VisualC++ 6.0 o .NET). Nel caso di scelta multipla Matlab™ chiede all'utente quale usare.

Per compilare il file MySource.m ed ottenere l'eseguibile MySource.exe:

```
mcc –m MySource.m
```

N.B.: il file MySource.m non deve essere uno script di Matlab bensì una function (vedi discussione riportata alla sezione: Struttura dei file del presente Tutorial).

Bibliografia

- Moler C., “Numerical Computing with MATLAB”, The Mathworks, (2004)
- http://www.mathworks.com/moler
- http://www.eece.maine.edu/mm/matweb.html
- http://spicerack.sr.unh.edu/~mathadm/tutorial/software/matlab/
- http://www.engin.umich.edu/group/ctm/basic/basic.html
- http://www.mines.utah.edu/gg_computer_seminar/matlab/matlab.html
- http://www.math.ufl.edu/help/matlab-tutorial/
- http://www.indiana.edu/~statmath/math/matlab/
- http://www.mathworks.com/support/tech-notes/1600/1622.html