Introduzione ai linguaggi di scripting nel calcolo

Piero Lanucara, Gabriele Mambrini

CASPUR

Via dei Tizii, 6*b* — 00185 Rome — Italy

http://www.caspur.it/

e-mail: {lanucara,mambrini}@caspur.it

Sommario:

- Introduzione
- Caratterizzazione dei tre linguaggi di scripting (Perl, Matlab e Python)
- Il problema del tuning e possibili soluzioni
- Conclusioni

Definizione di script

- Un programma, spesso breve e conciso, scritto in un linguaggio di alto livello (linguaggio di scripting).
- Linguaggi di scripting: Matlab, Perl, Python ma anche Tcl, Ruby, JavaScript, VisualBasic, ...

Con un linguaggio di scripting si può:

- incollare tra loro codici preesistenti
- processare files di testo molto grandi
- manipolare files e directory
- usufruire di operatori specializzati di alto livello
- creare applicativi di non grandi dimensioni e con un breve ciclo di sviluppo
- creare applicativi con una GUI integrata
- avere portabilità su piattaforme Unix, Windows e Macintosh
- avere un codice interpretato (assenza di compilazione/linking)
- ...

Perché non Fortran o C/C++?

- Codici più brevi.
- Sviluppo dell'applicativo molto più rapido.
- Nessuna dichiarazione di variabili ma verifica della consistenza a RunTime.
- Strumenti di alto livello per la gestione dei processi e il processamento di testi.
- Maggiore semplicità nella realizzazione di Interfacce grafiche (GUI) e Web.

La Preistoria

Unix si è evoluto come un ambiente di sviluppo altamente produttivo grazie alla combinazione di due strumenti alquanto differenti:

- un linguaggio standard per tasks di calcolo intensivi (il C)
- un ambiente di *shell* per incollare tra di loro i programmi C e formare nuove applicazioni.

Un esempio:

```
du -a $HOME |sort -rn |less
```

Il bisogno di strumenti software che estendessero queste funzionalità ha portato alla nascita dei linguaggi di scripting odierni.

La popolarità dei linguaggi di scripting è in crescita!

- negli anni 70 sviluppo di Matlab
- negli anni 90 sviluppo di Perl, Python, etc
- Y2K: i linguaggi di scripting sono stabili e supportati su tutte le maggiori piattaforme di calcolo
- Futuro : interesse esplosivo. Perché ?
  1. Riutilizzo del software.
  2. GUI.
  3. High Performance Script.

Codici più compatti

I linguaggi di scripting consentono di avere codici notevolmente più compatti rispetto ai corrispettivi Fortran, C, C++ etc.
Il motivo è da ricercarsi nei numerosi costrutti e strutture dati di alto livello supportati. Un esempio in Matlab:

```
>> a = rand(100);
>> imagesc(a);
>> colormap(hot);
>> axis square;
>> b = inv(a);
>> imagesc(b);
>> axis square;
```

Codici più compatti — 2

Vediamo un esempio non numerico. Supponiamo di dover importare un file di numeri reali dove ogni riga può contenerne un numero arbitrario:

```
1.1 9 5.2
1.762543E-02
0 0.01 0.001
9 3 7
```

La soluzione in Python:

```
F = open(filename, 'r')
n = F.read().split()
```

Regular Expressions

Sono molto utilizzate nel processamento di testi. Supponiamo di voler importare coppie di numeri complessi:

```
(-3, 1.4)        (4,2)
```

La soluzione in Perl è molto semplice sebbene non leggibilissima:

```
$s = "(-3,1.4)";
($re,$im) = $s =~ /\(\s*([^,]+)\s*,\s*([^,]+)\s*\)/;
```

Espressioni come:

```
\(\s*([^,]+)\s*,\s*([^,]+)\s*\)
```

costituiscono uno strumento potentissimo. Fare la stessa cosa con un linguaggio come il C o Fortran richiede maneggiare strutture a livello di tipi *character*.

Dichiarazione delle Variabili

I linguaggi “nativi” richiedono che ogni variabile sia espressamente dichiarata di un certo tipo (float, REAL*8, etc). Il compilatore utilizza queste informazioni per controllare che un certo algoritmo utilizzi il dato correttamente. Ad esempio in C:

```
double c; c = 5.2; #c can only hold doubles
c = "a string ..." # compiler error
```

Si dice in questo caso che il linguaggio è *staticamente tipato*. Vantaggi e svantaggi:

\+ diminuisce la possibilità di *bugs* nel codice e la programmazione risulta più corretta.

– la flessibilità diminuisce. Questo fatto può portare ad una diminuzione della produttività laddove vi sia un forte riutilizzo dei codici.

Dichiarazione delle Variabili — 2

Dei linguaggi di scripting si dice che sono *dinamicamente tipati* poiché le variabili non sono dichiarate di un qualche tipo e non ci sono restrizioni *a priori* su come queste variabili saranno combinate. Un esempio in Python:

```
c = 1 # c is an integer
c = [1,2,3]  # c is a list
```

Vantaggi e svantaggi:

+ grande flessibilità.
– aumenta la possibilità di effetti indesiderati dovuti a errata battitura, errori di sintassi, etc.

Dichiarazione delle Variabili — 3

All'interno dei linguaggi di scripting possiamo comunque suddividere ulteriormente in *debolmente tipati* (il Perl e Matlab) e *fortemente tipati* come il Python. Ad esempio in Perl è perfettamente lecita l'espressione:

```
$b = '1.2';
$c = 5*$b; # implicit type conversion '1.2' -> 1.2
```

mentre in Python:

```
r = sys.argv[1]
s = math.sin(r) # sine of a string...
Traceback (innermost last):
  File "hw.py", line 10, in ?
    s = math.sin(r)
TypeError: illegal argument type for built-in operation
```

Prototipazione rapida

- Calcolo interattivo: l'utente può digitare un comando e subito vederne l'output:

```
>> A=rand(100);max(eig(A))

ans =

   50.5051
```

- I comandi possono essere richiamati ed editati immediatamente con il *text editor* preferito.
- Una volta che la fase di prototipazione è terminata si può salvare il tutto in un file testo e mandarlo in esecuzione immediatamente.

Creazione di codice a RunTime

- Poiché lo script è interpretato è possibile generare codice mentre lo script è in esecuzione.
- Vediamo un esempio di file di input avente la seguente sintassi:

```
a = 1.2
no of iterations = 100
solution strategy = 'implicit'
c1 = 0
c2 = 0.1
A = 4
c3 = StringFunction('A*sin(x)')
```

- Come leggere questo file assegnando a ciascuna variabile il valore specificato?

Creazione di codice a RunTime — 2

- La risposta è in questo semplice script Python:

```
file = open('inputfile.dat', 'r')
for line in file:
    # replace blanks on left-hand side of = by _
    variable, value = line.split('=').strip()
    variable = re.sub(' ','_', variable)
    exec(' = '.join(variable, value))
```

- Le variabili c1, c2, c3, etc sono create a RunTime e con i valori dati dal file in input.
- Non è possibile fare la stessa cosa in Fortran, C, C++, ...

Oltre il *Number Crunching*

Molti sviluppatori di software hanno compreso che una gran parte del loro lavoro non riguarda l'implementazione di routines di calcolo numerico intensivo.

Spesso occorre:

- muovere dati tra differenti pacchetti
- convertire dati tra differenti formati
- estrarre dati utili da un file testo
- amministrare files e directory relativi ad uno specifico run
- analizzare e visualizzare i risultati ottenuti

Oltre il *Number Crunching* — 2

Inoltre, molti programmatori C e Fortran vorrebbero usare i propri codici testati ed efficienti in ambienti più moderni senza migrarli a C++ o Java. Dato un cosiddetto *legacy code* si vorrebbe:

- incapsularlo in librerie (C, Fortran, ...)
- dotarlo di una GUI
- mandarlo in esecuzione
- visualizzare i risultati ottenuti

Questi task sono difficilmente gestibili da linguaggi di programmazione come il C o il Fortran. Sono viceversa facilmente eseguiti tramite un qualsiasi linguaggio di scripting.

Oltre il *Number Crunching* — 3

In generale, si possono configurare due modalità di comunicazione tra le varie parti dell'applicativo globale:

- gli eseguibili comunicano attraverso files di dati. È il modo più semplice e naturale. Un ottimo esempio che utilizza l'ambiente Tcl–Tk è la libreria HJPACK (*www.caspur.it/hjpack/*) interamente sviluppata a CASPUR da M. Rorro.

- Si utilizzano chiamate a funzioni e puntatori per gestire il trasferimento dei dati tra le applicazioni. È il modo più efficiente sebbene sia meno intuitivo.

GUI

- Le GUI (Graphical User Interface) rappresentano ormai una parte cruciale di molti pacchetti software.
- Programmare una GUI in C o Fortran risulta estremamente noioso oltre che difficile per lo sviluppatore software medio.
- Molti linguaggi di scripting consentono di costruire rapidamente delle GUI o di accoppiare una GUI alla propria applicazione.
- Di fatto, la natura stessa dei linguaggi di scripting incoraggia a scrivere applicazioni indipendenti incollate tramite brevi scripts piuttosto che applicazioni *stand-alone* molto pesanti ed equipaggiate di GUI complicate.

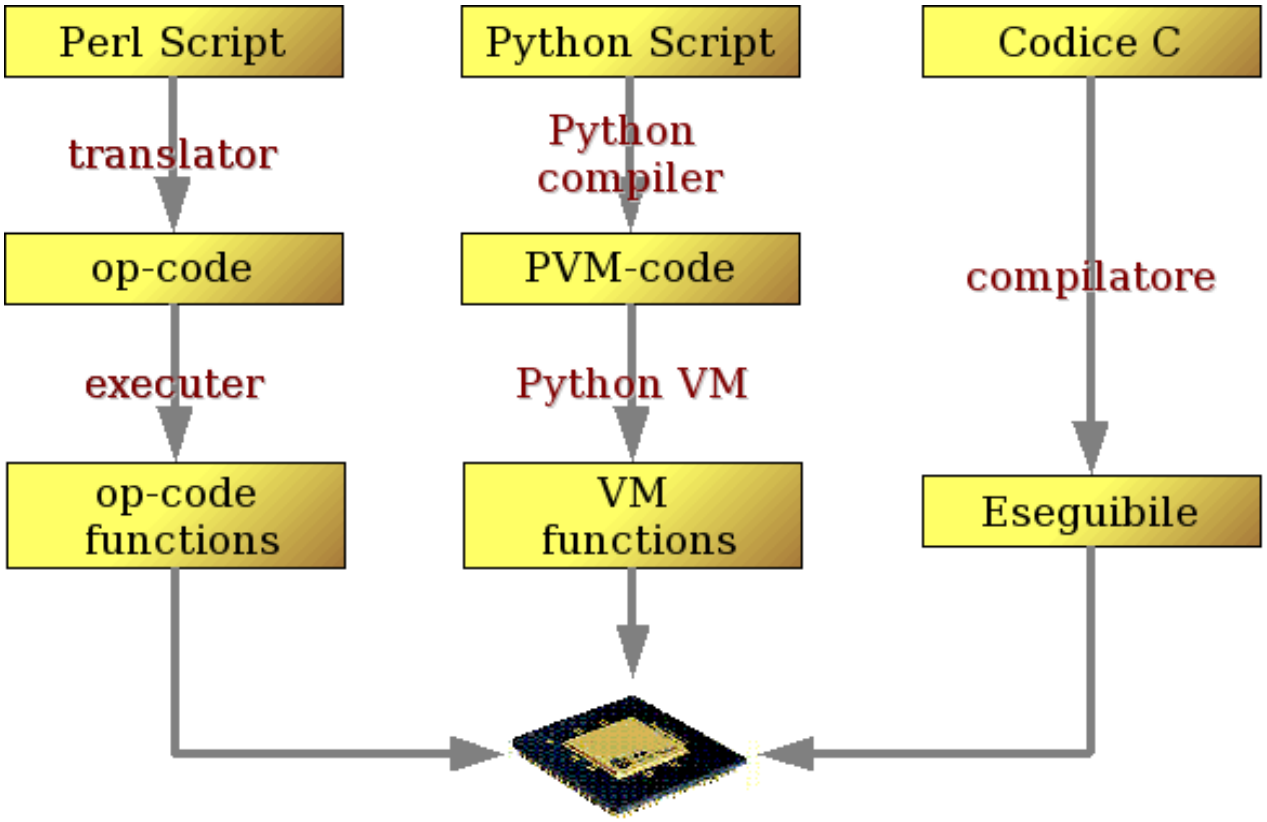
Applicazioni Web

- Una classe particolare di GUI dove l'utente inserisce dei dati e prende indietro dei risultati ha assunto particolare importanza in questi anni.
- Linguaggi come il Perl, Python ma soprattutto PHP sono diventati sempre più popolari nello sviluppo di programmi sul lato (web-)server poiché consentono un rapido sviluppo dell'applicativo Web.
- Questo tipo di programmi prende il nome di *server side* scripts.

La potenza di Unix su Windows

- Molti ricercatori nascono naturalmente come programmatori in ambiente Unix. Muoversi in ambiente Microsoft risulta allora particolarmente stressante.
- I linguaggi di scripting facilitano questo processo in quanto sono per definizione *cross-platform* e funzionano allo stesso modo anche su Windows/Macintosh.

Il problema dell'efficienza



Il problema dell'efficienza — 2

- I linguaggi di scripting vengono trasformati dall'interprete in bytecode e poi eseguiti.
- Viceversa linguaggi come il C, Fortran, etc (con l'eccezione di Java) vengono tradotti in istruzioni macchina fortemente dipendenti dall'hardware.
- I linguaggi di scripting possono soffrire di mancanza di prestazioni, in quanto inducono *overhead* e non consentono di usare l'hw al meglio.
- Tuttavia in alcuni casi i linguaggi di scripting ottengono prestazioni molto elevate: un ottimo esempio riguarda il processamento di testi, che sfrutta implementazioni ottimizzate.
- In ultima analisi si possono utilizzare tecniche per migliorare le prestazioni dei linguaggi di scripting.

Il problema dell'efficienza — 3

- Usare differenti linguaggi per task differenti all'interno della propria applicazione può rivelarsi una strategia vincente.
- I linguaggi di scripting sono generalmente implementati in C e quindi risulta abbastanza semplice estendere le loro potenzialità con funzioni scritte in questo linguaggio. In ogni caso risultano integrabili con Fortran e C++ senza grossi problemi.
- Si utilizzerà un linguaggio di scripting per creare un ambiente user-friendly con interattività, sintassi chiara e compatta, visualizzazione etc, migrando le parti computazionalmente più lente a C, Fortran o C++.

Il linguaggio Matlab

Introduzione

- MATLAB è un linguaggio di alto livello per il calcolo scientifico. Esso integra in un solo ambiente il calcolo, la visualizzazione e la programmazione in un ambiente *user–friendly*.
- L'elemento base di MATLAB è l'*array*. Ciò consente la risoluzione di tutti quei problemi di calcolo scientifico aventi formulazioni vettoriali e matriciali attraverso algoritmi molto più compatti rispetto a quelli scritti in linguaggi come C, Fortran, etc.
- Il nome MATLAB deriva da (MATrix LABoratory) e si è sviluppato a partire dagli ambienti numerici LINPACK ed EISPACK negli anni 70.
- MATLAB si è costantemente evoluto negli anni anche grazie ai contributi dati da utenti di tutto il mondo e all'utilizzo dei *toolbox* per risolvere particolari categorie di problemi.

Calcolo Matriciale

Importiamo una matrice in MATLAB:

```
>> A=magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

È molto semplice verificare che si tratta di una matrice *magic square*.

Somma sulle colonne:

```
>> sum(A)
ans =
    34    34    34    34
```

Calcolo Matriciale — 2

Anche la somma sulle righe:

```
>> sum(A')
ans =
    34    34    34    34
```

Nondimeno la somma della diagonale principale:

```
>> sum(diag(A))
ans =
    34
```

e la somma dell'antidiagonale utilizzando la funzione MATLAB *fliplr*:

```
>> sum(diag(fliplr(A)))
ans =
    34
```

Variabili

MATLAB non richiede alcuna dichiarazione del tipo della variabile. Quando MATLAB incontra un nome nuovo crea automaticamente la variabile e alloca il giusto:

```
>> num_students = 25;
>> pi_greco=pi;
>> name_student='Anna';
>> whos
  Name                Size                     Bytes  Class

  name_student        1x4                          8  char array
  num_students        1x1                          8  double array
  pi_greco            1x1                          8  double array
Grand total is 6 elements using 24 bytes
```

Allocazione Dinamica in MATLAB

Supponiamo di aver definito la matrice X mediante il comando:

```
X = eye(4);
```

L'assegnazione:

```
X(4,5) = 17;
X =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1    17
```

crea una variabile X *sufficientemente* larga riallocando la memoria necessaria affinche l'assegnazione abbia senso.

Operazioni

MATLAB lascia massima flessibilità al programmatore. La stessa operazione può riferirsi a uno scalare, una matrice, etc:

```
>> A=magic(4);
>> B=1;
>> c=A*A
c =
   345   257   281   273
   257   313   305   281
   281   305   313   257
   273   281   257   345
>> c=B*B
c =
     1
```

Scripts e Funzioni

MATLAB può essere utilizzato oltre che come ambiente computazionale interattivo come un vero e proprio linguaggio di programmazione.
Files che contengono codice MATLAB sono chiamati *M-files* e possono essere creati utilizzando un qualunque *text editor*.
Ci sono due grosse categorie di *M-files*:

- *Scripts*: non hanno argomenti in entrata o in uscita. Operano su dati nel *workspace*.
- *Funzioni*: possono avere argomenti in entrata e in uscita. Una variabile interna è *locale* alla funzione.

Quando si richiama uno script, MATLAB esegue semplicemente i comandi presenti nel file.

Scripts

Sebbene gli script non forniscano dati di output, qualsiasi variabile da loro creata rimane nel *workspace*, per essere usata in calcoli susseguenti. Inoltre, gli script possono produrre dei grafici, usando funzioni di *plot*.

```
% Determinazione del rango della magic squares
r = zeros(1,32);
for n = 3:32
r(n) = rank(magic(n));
end
r
bar(r)
```

Se vogliamo eseguire lo script diamo da dentro MATLAB il comando *magicrank*.

Funzioni

Le funzioni sono *M-files* che possono accettare argomenti in input e forniscono argomenti in output.
Il nome dell'*M-file* e della funzione deve essere lo stesso.
Le Funzioni operano su variabili definite nel proprio *workspace*, separato dal *workspace* a cui si accede all'ingresso di MATLAB, cioè le variabili usate all'interno della funzione sono locali. Un esempio di funzione:

```
function r = rank(A,tol)
s = svd(A);
if nargin==1
tol = max(size(A)) * max(s) * eps;
end
r = sum(s > tol);
```

Interfacce a funzioni flessibili

Una delle caratteristiche di MATLAB è quella di permettere interfacce a funzioni flessibili. Pertanto:

```
x=0:0.1:pi;
y=sin(x);
plot(x,y)
```

realizza un *plot* standard mentre l'istruzione più complessa:

```
plot(x,y,'LineWidth',10,'Color',[.6 0 0])
```

realizza un controllo più fine sul tipo di grafico, sul colore, etc.

Strutture di controllo del flusso

La prima struttura presente è comune a tutti i linguaggi di programmazione.
In MATLAB il seguente costrutto:

```
if rem(n,2) ~= 0
M = odd_magic(n)
elseif rem(n,4) ~= 0
M = single_even_magic(n)
else
M = double_even_magic(n)
end
```

valuta rispettivamente il caso in cui n sia dispari, pari ma non divisibile per quattro e divisibile per quattro. Si noti l'utilizzo della funzione *rem* che calcola il resto della divisione.
Analoga struttura compare nel costrutto *switch–case*.

Strutture di controllo del flusso — 2

Molto importante è il costrutto *for*:

```
for n = 3:32
   r(n) = rank(magic(n));
end
```

Andrebbe utilizzato solamente dove non è possibile *vettorizzare* il codice. Una sorta di ciclo *for* infinito si ha attraverso il costrutto *while–end*. Il criterio di uscita viene dato mediante una condizione logica opportuna:

```
a = 0; fa = -Inf; b = 3; fb = Inf;
 while b-a > eps*b
     x = (a+b)/2;
     fx = x^3-2*x-5;
 end
```

Valutazione di stringhe

Questa funzionalità aumenta notevolmente la potenza e la flessibilità dell'ambiente MATLAB.

La funzione *eval* valuta una stringa che contiene un'espressione, istruzione o una chiamata a funzione. La seguente chiamata può essere utilizzata per generare una matrice di *Hilbert* di ordine n:

```
t = '1/(m + n - 1)';
for m = 1:k
    for n = 1:k
        a(m,n) = eval(t);
    end
end
```

Valutazione di stringhe — 2

Si possono concatenare stringhe per creare espressioni sempre più complesse. L'espressione seguente crea 10 variabili P1, P2, ...P10:

```
for n = 1:10
    eval(['P', int2str(n), '= n .^ 2'])
end
```

La funzione *feval* viene usata per valutare funzioni piuttosto che espressioni MATLAB. Vediamo un esempio:

```
x=0:1:pi;
fun = @sin;
feval(fun, x)
ans =
         0    0.8415    0.9093    0.1411
```

Hashtable in Matlab

- Un *Hashtable* (o dizionario) è una struttura dati in cui è possibile definire degli *array* indicizzati in modo arbitrario.
- È possibile utilizzare *Hashtable* grazie al sito degli utenti MATLAB assolutamente gratuito e da cui è possibile scaricare *M–files*. Il sito è al link:

  `http://www.mathworks.com/matlabcentral/`

- L'unica limitazione è che il campo degli indici (*Keys*) deve essere una stringa mentre i dati possono essere di qualunque tipo.

Hashtable in Matlab — 2

Un esempio di *Hashtable*:

```
>> hash = hashtable;
>> hash = put(hash,'random numbers',rand(5));
>> hash = put(hash,'b','abcdefg...');
```

La struttura così definita contiene una matrice 5×5 di numeri *random* e un campo stringa.

Hashtable in Matlab — 3

Con *put* e *get* metto e prendo dati nella (dalla) *Hashtable*:

```
>> get(hash,'random numbers')
ans =
    0.4741    0.5165    0.2452    0.5074    0.3589
    0.2572    0.2126    0.0396    0.2919    0.7385
    0.3252    0.4850    0.8854    0.1834    0.0861
    0.0845    0.4113    0.0348    0.4514    0.4469
    0.6618    0.2088    0.6223    0.5771    0.2273
```

Con *keys* e *values* si estraggono rispettivamente le chiavi e i dati contenuti nella *Hashtable*:

```
>> k = keys(hash)
k = 'random numbers'    'b'
```

Hashtable in Matlab — 4

```
>> v = values(hash)
v =
    [5x5 double]    'abcdefg...'
```

Con *remove* elimino elementi dalla *Hashtable*. Posso quindi definire una nuova *Hashtable* utilizzando le funzioni precedenti:

```
>> hash = remove(hash,'random numbers');
>> k = keys(hash);
>> v = values(hash);
>> newhash = hashtable(k,v)
newhash =
        HashTable
        Elements:
    'b'    'abcdefg...'
```

GUI in Matlab

- Utilizzare GUI in MATLAB può essere conveniente per avere un ambiente di sviluppo quasi identico su piattaforme Unix (Linux) o Windows. L'unica avvertenza riguarda la necessità di ricompilare codici C se presenti nell'applicazione.
- Lo script MATLAB *guide* (GUI Design Environment) consente di inizializzare l'ambiente GUI, specificare le dimensioni della finestra, aggiungere le componenti (pannello di lavoro, bottoni, etc) e proprietà.
- Il codice generato viene salvato in un *M–file* che contiene un *template* con le definizioni dei vari *callbacks*.
- L'utente deve, a questo punto, aggiungere codice per rendere i *callbacks* operativi.

Il linguaggio Perl

Introduzione

- Perl è un acronimo “Practical Extraction and Report Language”.
- È un linguaggio interpretato ottimizzato per manipolazione di stringhe, I/O e gestione di sistema.
- La sintassi è ispirata al C ma incorpora caratteristiche da Bourne shell, csh, awk, sed, grep.
- Deve la sua popolarità soprattutto all’uso in ambito sistemistico e alla creazione di script CGI.
- Motto del Perl: TMTOWTDY.
- La versione attuale di Perl è la 5.8. Esistono poi le distribuzioni per Windows, la più nota delle quali è *ActivePerl*.

Un primo esempio

```
#!/usr/bin/perl -w
print "Hello, World!\n";
```

- La prima riga viene usata in ambiente Unix per rendere lo script eseguibile.
- Non c'è "main".
- Come per la shell i commenti cominciano per `#` e continuano sino a fine riga.
- I doppi apici definiscono una stringa.
- Tutte le istruzioni terminano con `;`

Variabili scalari

- Gli scalari possono essere stringhe, interi o numeri in virgola mobile.

```
$numero = 1974;
$stringa = "Hello, World\n";
```

- Gli scalari possono essere usati in vari modi:

```
print $stringa;
print "La stringa e' $animal\n";
print "The square of $a is ", $a * $a, "\n";
```

- C'è uno scalare *magico* `$_` che viene usato come argomento di default in numerose funzioni ed è assegnato automaticamente in alcuni tipi di loop

```
print;          # prints contents of $_ by default
```

Stringa o numero? Una questione di contesto

In Perl vige la massima flessibilità. La conversione è risolta automaticamente dall'interprete:

```
$a = "100";
$b = 200;
```

il primo scalare è definito come stringa il secondo come intero. La somma:

```
$c = $a + $b;
```

otterrà correttamente 300 come risultato. Allo stesso modo:

```
$c = $a . $b;
```

darà come risultato la stringa 100200.

Array

- Un *array* in Perl rappresenta una lista di valori:

```
@nomi = ('Michela', 'Elena', 'Luca', 'Elisa');
```

- L'inizializzazione è del tutto facoltativa.
- Gestione degli elementi dell'array:

```
print $nomi[0];
$nomi[2] = 'Daniele';
```

- In molti casi è utile conoscere il numero di elementi di un array:

```
print scalar(@nomi);
```

- Si possono ottenere più elementi per volta

```
@nomi[0,1];                    # ('Michela', 'Elena');
@animals[0..2];                # ('Michela', 'Elena', 'Luca')
@animals[1..$#animals];        # tutti tranne il primo
```

- In Perl il ridimensionamento degli array avviene automaticamente:

```
$nomi[10] = 'Alessia';
print join(',', @nomi),"\n";
```

stampa `Michela,Elena,Luca,Elisa,,,,,,,Alessia`.

- Si possono fare cose interessanti con le liste:

```
my @sorted    = sort @animals;
my @backwards = reverse @numbers;
my @squared = map { $_ * $_ } @numbers;
```

Array ed Hash

- Un hash rappresenta un insieme di coppie chiave/valore.
- È una struttura simile ad un array: al posto degli indici sono però utilizzate stringhe (le *chiavi*)

```
my %fruit_color = (
                    apple  => "red",
                    banana => "yellow",
                  );
$fruit_color{"apple"};           # gives "red"
$fruit_color{"blackberry"} = "black";
```

- È possibile ottenere un array contente tutte le chiavi oppure i valori di un *Hash* tramite le funzioni *keys()* e *values()*:

```
@k = keys %articolo;  @v = values %articolo;
```

Strutture di controllo

```
if ( condition ) {
    ...
} elsif ( other condition ) {
    ...
} else {
   ...
}
```

La condizione può essere qualunque espressione. Esiste anche una versione più “tipica” per fare piccole strutture if

```
if ($zippy) { print "Yow!"; }
# oppure
print "Yow!" if $zippy;
```

Strutture iterative — for e while

La struttura iterativa più nota prevede l'impiego dell'istruzione *for*:

```
for ($i = 0; $i <=3; $i++) {
  print "2 x $i = ".(2*$i)."\n";
}
```

Questo scampolo di codice esegue 4 volte la riga contenuta tra parentesi graffe. Il *while* è un'altra struttura iterativa fondamentale:

```
$i = 0;
while ($i <=3) {
  print "2 x $i = ".(2*$i)."\n";
  $i++;
}
```

Strutture iterative — foreach

foreach consente di scandire un array elemento per elemento.

```
@luoghi = ('Roma', 'Berlino', 'New York', 'Melbourne');
foreach $i (sort @luoghi) {
        print "$i\n";
}
# oppure
# foreach (sort @luoghi) { print "$_\n"; }
# print "$_\n" foreach (sort @luoghi);
```

causerà la stampa di tutte le città ordinate alfabeticamente senza che l'array *@luoghi* venga modificato.

Strutture dati multidimensionali

Utilizzando il concetto di *reference* è possibile usare hash e liste all'interno di un hash o una lista creando strutture complesse come array multidimensionali, hash di hash, etc.

```
@arr = (
 [ "perl", "rocks" ],
 [ "oh", "yeah" ],
 [ "hooray", "whoop!"],
);
print "$arr[1][1]\n";
```

```
% HoA = (
       flintstones        => [ "fred",   "barney" ],
       jetsons            => [ "george", "jane",  "elroy" ],
       simpsons           => [ "homer",  "marge", "bart" ],
);
@members = $HoA{'simpsons'};
# append new members to an existing family
push @{ $HoA{"flintstones"} }, "wilma", "betty";
```

Regular Expression in Perl

Le *regular expression* sono un potentissimo strumento per la manipolazione di dati. Con una *regular expression* è possibile sintetizzare in una sola riga di Perl ciò che ne potrebbe richiedere svariate decine.

```
if ($a =~ m/Anna/) {
    . . .
}
```

Controlla se $a contiene da qualche parte il nome *Anna*. L'operatore che lega la variabile all'espressione è un uguale seguito da una tilde.

Regular Expression in Perl

Benchè al primo impatto le *regular expression* risultino assai ostiche, sono uno strumento potentissimo per il programmatore. L'espressione complicatissima:

```
/^([\w\-\+\.]+)@([\w\-\+\.]+).([\w\-\+\.]+)$/
```

controlla la validità sintattica di un indirizzo di posta elettronica.
L'elemento comune è il *pattern*:

```
[\w\-\+\.]+
```

le parentesi quadre definiscono la classe di caratteri composta dai caratteri alfanumerici (incluso l'underscore), il segno meno, il segno più e il punto; il più finale indica "una o più occorrenze".

Esempio: word count

```
$filename = shift @ARGV;
open FH, $filename;
while ( $line = <FH> ) {
        $n_lines++;
        $line =~ s/^\s+//;
        @words = split /\s+/, $line;
        $n_words += @words;
}
print "$n_lines $n_words\n";
close FH;
```

Il linguaggio Python

Introduzione

- Python è un linguaggio di programmazione di alto livello, interpretato, orientato agli oggetti e con una semantica dinamica.
- Può essere utilizzato o come linguaggio di scripting ma anche come collante per connettere insieme componenti esistenti (magari scritti in linguaggi differenti).
- Python supporta moduli e pacchetti incoraggiando la programmazione modulare ed il riutilizzo del codice.
- In definitiva è molto apprezzato per la prototipazione rapida di grosse applicazioni.

Un semplice esempio

Consideriamo il seguente script Python:

```
#!/usr/bin/env python
import sys, math        # load system and math module
r = float(sys.argv[1]) # extract the 1st command-line arg.
s = math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

Mandando in esecuzione lo script si ottiene:

```
python hw.py 1.4
Hello, World! sin(1.4)=0.985449729988
```

Un semplice esempio

Possiamo anche dare il comando:

```
./hw.py 1.4
Hello, World! sin(1.4)=0.985449729988
```

in questo caso la prima linea dello script viene utilizzata per specificare l'interprete.
Anche questo semplice script deve caricare dei moduli:

```
import sys, math
```

e pertanto numeri e stringhe vengono trattati in modo differente:

```
r = sys.argv[1] # r is a string
s = math.sin(float(r)) # sin expects number, not string r
```

Exception handling

Python fornisce utili informazioni in caso di un errore di input da parte del programmatore:

```
./hw.py
Traceback (innermost last):
  File "./hw.py", line 3, in ?
    r = float(sys.argv[1]) # extract the 1st command-line arg.
IndexError: list index out of range
```

In questo caso non abbiamo fornito il reale in input. Python termina l'esecuzione dello script con un messaggio di errore informativo.
C'e anche la possibilità di gestire manualmente gli errori con i comandi *try* ed *except*.

Exception handling

Lo script precedente diventa:

```
#!/usr/bin/env python
import sys, math       # load system and math module
try:
    iarg = sys.argv[1];
except:
    print "Usage:",sys.argv[0], "input"; sys.exit(1)
r = float(sys.argv[1]) # extract the 1st command-line arg.
s = math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

Un altro esempio

Python consente di scrivere programmi molto compatti e di facile lettura. Tipicamente i programmi scritti in Python sono molto più brevi degli equivalenti in C o Fortran, per numerose ragioni tra cui:

- operazioni complesse possono essere espresse in una singola istruzione.
- le istruzioni sono raggruppate tramite indentazione (niente parentesi).
- non è necessario al solito dichiarare variabili e argomenti.

Un altro esempio

Vediamo un esempio di script Python in cui si leggono dati da un file a due colonne, si trasformano secondo una certa legge e si scrivono su un nuovo file.

```
#!/usr/bin/env python
import sys, math
try:
    infilename = sys.argv[1];  outfilename = sys.argv[2]
except:
    print "Usage:",sys.argv[0], "infile outfile"; sys.exit(1)
ifile = open( infilename, 'r')  # open file for reading
ofile = open(outfilename, 'w')  # open file for writing
```

Un altro esempio

```
def myfunc(y):
    if y >= 0.0:  return y**5*math.exp(-y)
    else:         return 0.0
# read ifile line by line and write out transformed values:
for line in ifile:
    pair = line.split()
    x = float(pair[0]); y = float(pair[1])
    fy = myfunc(y)  # transform y value
    ofile.write('%g  %12.5e\n' % (x,fy))
ifile.close();  ofile.close()
```

Si noti l'indentazione del blocco di lettura che elimina di fatto le parentesi.

Liste e Tuple in Python

L'oggetto *lista* in Python definisce una collezione di numeri, stringhe e qualsiasi altra struttura dati:

```
arglist = [myarg1, 'displacement', "tmp.ps"]
```

Una *tupla* si definisce invece:

```
(item1, item2, ...)
```

Non si può cambiare il contenuto di una *tupla*:

```
words = ('tuple', 'rhymes with', 'couple')
words[1] = 'and' # illegal - Python issues an error message
```

mentre una lista si comporta come un *array* in C o Fortran.

Scorrere una lista

L'istruzione *for* in Python presenta una flessibilità sconosciuta a linguaggi come il C o Fortran. Infatti il *for* compie un'iterazione sugli elementi di una lista o *tupla* nell'ordine un cui appaiono nella sequenza. Ad esempio:

```
>>> # Misura la lunghezza di alcune stringhe:
... a = ['gatto', 'finestra', 'defenestrare']
>>> for x in a:
...     print x, len(x)
gatto 5
finestra 8
defenestrare 12
>>> for x in range(1,3):
...     print x
1
2
```

List Comprehensions

Sono un modo conciso per creare liste senza usare map, filter e lambda.

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Hash

Un *Hash* in Python viene definito esattamente come negli altri linguaggi con l'avvertenza che la *key* può essere un oggetto qualsivoglia (stringhe, numeri e *tuple*) ma non *liste*:

```
cmlargs = {}              # initialize as empty dictionary
cmlargs['m'] = 1.2        # add 'm' key and its value
cmlargs['tstop'] = 6.0
```

A questo punto possiamo estrarre *keys* e *values*:

```
>>> cmlargs.keys()
['tstop', 'm']
>>> cmlargs.values()
[6.0, 1.2]
```

Funzioni

Una tipica funzione Python può essere definita come:

```
def function_name(arg1, arg2, arg3):
    # statements
    return something
```

Python estende notevolmente il concetto di funzione, consentendo di ritornare qualunque struttura dati (eventualmente *none*) e l'uso di argomenti *keywords*:

Funzioni — 2

```
def mkdir(dir, mode=0777, remove=True, chdir=True):
    if os.path.isdir(dir):
       if remove:
          shutil.rmtree(dir)
       elif :
          return False # did not make a new directory
    os.mkdir(dir, mode)
    if chdir: os.chdir(dir)
    return True          # made a new directory
```

Funzioni — 3

Nell'esempio *dir* è un argomento posizionale mentre *mode*, *remove* e *chdir* sono argomenti *keywords* con valori assegnati di default. Dando il comando:`mkdir('tmp1')` *tmp1* se esiste viene rimossa, quindi creata e la *working directory* diviene *tmp1*.
Si possono anche definire un numero variabile di argomenti, funzioni in oggetti Python, *Lambda Functions*, etc.

Progettazione di GUI

- Python è un ottimo strumento per la prototipazione rapida di applicazioni. Un'applicazione particolarmente pesante avrà grande beneficio dall'introduzione di una GUI il più possibile *user–friendly*.
- Python utilizza la librerie *Tk* attraverso un'interfaccia chiamata *Tkinter*. Vediamo un semplice esempio (`sin-gui.py`).
- Possiamo scrivere GUI più complesse implementandole come classi oppure attaccare delle GUI ad uno script Python.
- L'obiettivo è avere uno script che prende in input dei dati attraverso una GUI, li passa alla simulazione e visualizza i risultati.

Il problema del *tuning* e possibili soluzioni

Il problema dell'efficienza

- Abbiamo visto che i linguaggi di scripting vengono prima compilati in un *byte-code* indipendente dall'hardware che poi viene *interpretato*.
- Viceversa linguaggi come il C, Fortran traducono il codice in istruzioni macchina fortemente dipendenti dall'hardware.
- Questo fatto penalizza quasi sempre le prestazioni dei linguaggi di scripting sebbene in qualche caso si possano avere risultati addirittura migliori (ad esempio nella processazione di testi).
- In ogni caso è sempre possibile utilizzare delle tecniche *ad-hoc* per ciascun linguaggio di scripting che consentano di migliorare l'efficienza dei codici mantenendone però la flessibilità, semplicità etc.

Il problema dell'efficienza — 2

Le tecniche che verranno mostrate ricadono essenzialmente in due grandi categorie:

- ottimizzazione del codice utilizzando tecniche proprie del linguaggio di scripting utilizzato.
- utilizzo del cosiddetto *mixed language programming* in cui il linguaggio di scripting viene integrato con Fortran, C, C++, Java etc per risolvere in modo opportuno possibili *bottlenecks* utilizzando il linguaggio appropriato.

Scrivere codice MATLAB veloce

MATLAB offre numerose possibilità per velocizzare i suoi codici.
Chiaramente queste tecniche vanno utilizzate con cautela:

- un codice estremamente ottimizzato risulta spesso illegibile
- non conviene ottimizzare una parte se questa pesa molto poco all'interno dell'applicazione

Il *Profiler*

Questo *tool* aiuta a determinare i *bottlenecks* nel codice. Vediamo un semplice esempio:

```
function result = example1(Count)
        for k=1:Count
            result(k) = sin(k/50);
            if result(k) < -0.9
               result(k) = gammaln(k);
            end
        end
```

Possiamo mandare in esecuzione la *function* dando il parametro di input (*Count*):

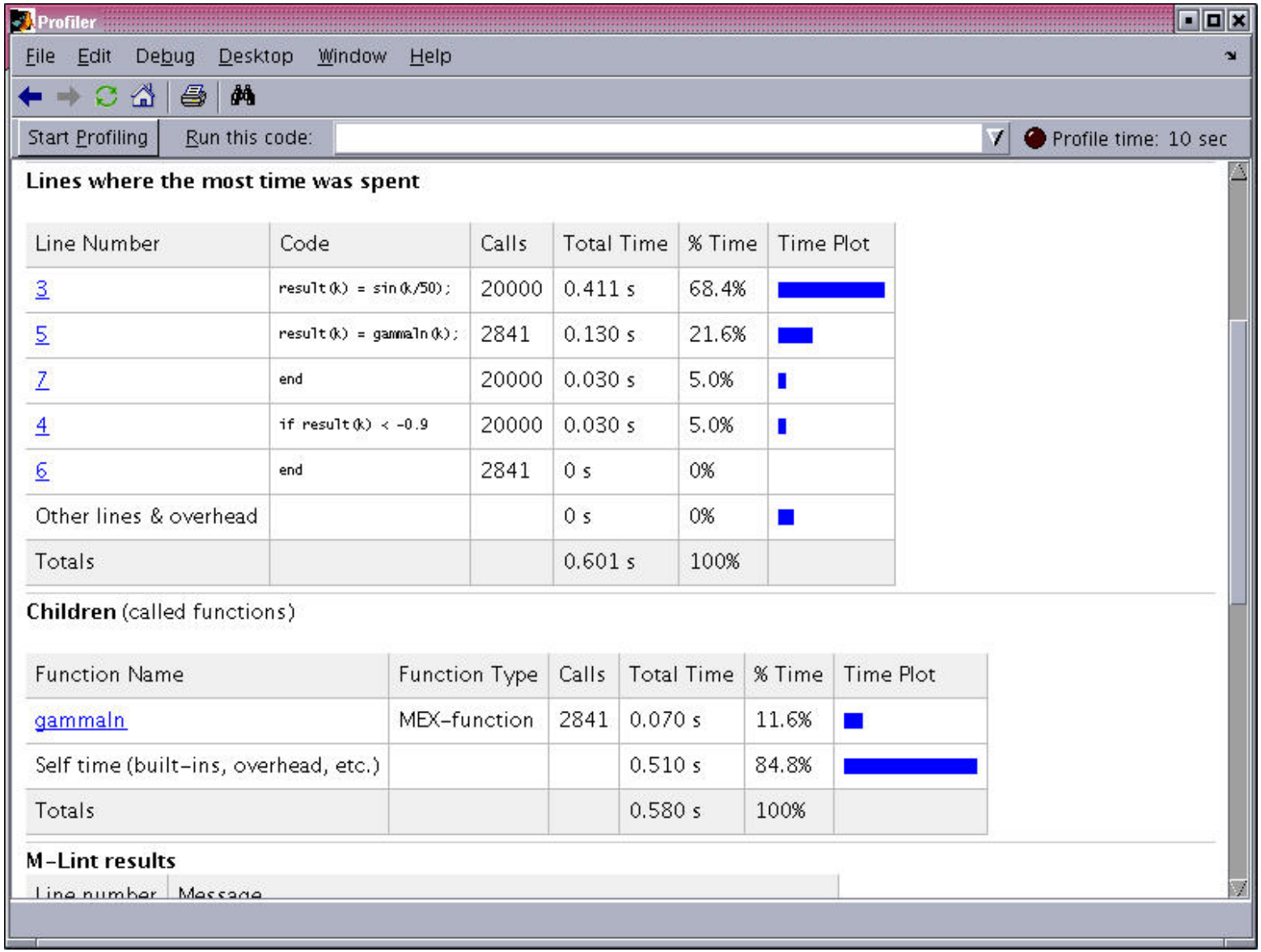
```
>> tic;example1(5000);toc
```

Il *Profiler* — 2

Per analizzare l'efficienza della *function* abilitiamo il *Profiler*:

```
>> profile clear
>> profile on
>> tic;example1(20000);toc
Elapsed time is 0.593666 seconds.
>> profreport('example1')
```

Il *Profiler* genera un file HTML e lancia una finestra di *report* della frazione di tempo di esecuzione speso da ciascuna *function*.
Si può andare maggiormente in dettaglio istanziando la *function* con un semplice *mouse–click*:


Profiler
File Edit Debug Desktop Window Help
Start Profiling
Run this code:
Profile time: 10 sec
Lines where the most time was spent
Line Number | Code | Calls | Total Time | % Time | Time Plot
3 | result(k) = sin(k/50); | 20000 | 0.411 s | 68.4%
5 | result(k) = gammaln(k); | 2841 | 0.130 s | 21.6%
7 | end | 20000 | 0.030 s | 5.0%
4 | if result(k) < -0.9 | 20000 | 0.030 s | 5.0%
6 | end | 2841 | 0 s | 0%
Other lines & overhead | | | 0 s | 0%
Totals | | | 0.601 s | 100%
Children (called functions)
Function Name | Function Type | Calls | Total Time | % Time | Time Plot
gammaln | MEX-function | 2841 | 0.070 s | 11.6%
Self time (built-ins, overhead, etc.) | | | 0.510 s | 84.8%
Totals | | | 0.580 s | 100%
M-Lint results
Line number | Message

Preallocazione

Una delle caratteristiche di MATLAB e di tutti i linguaggi di scripting in generale è l'allocazione della memoria dinamica:

```
>> a=2
a =
     2
>> a(2,6)=1
a =

     2     0     0     0     0     0
     0     0     0     0     0     1
```

Internamente la memoria allocata dalla matrice deve essere riallocata. Se quest'operazione avviene all'interno di un ciclo la velocità di esecuzione può risentirne.

Preallocazione — 2

Per evitare questo problema è buona norma preallocare la memoria richiesta dalla matrice al suo valore massimo. Vediamolo con un semplice esempio:

```
a(1)=1;
b(1)=0;
for k=2:20000
a(k)=0.99803*a(k-1)-0.06279*b(k-1);
b(k)=0.06279*a(k-1)+0.99803*b(k-1);
end
```

Mandandolo in esecuzione:

```
>> tic;example;toc
Elapsed time is 0.923572 seconds.
```

Preallocazione — 3

Cambiamo lo script utilizzando la function *zeros* di MATLAB ed otteniamo un notevole *speed–up* nelle prestazioni:

```
a=zeros(1,20000);
b=zeros(1,20000);
a(1)=1;
b(1)=0;
for k=2:20000
a(k)=0.99803*a(k-1)-0.06279*b(k-1);
b(k)=0.06279*a(k-1)+0.99803*b(k-1);
end
```

```
>> tic;example;toc
Elapsed time is 0.001954 seconds.
```

Vettorizzazione

Si dice che un calcolo è vettorizzato quando si eseguono delle operazioni sull'intero *array* piuttosto che elemento per elemento.
La vettorizzazione sfrutta appieno le potenzialità di MATLAB poiché molte funzioni MATLAB sono già vettorizzate:

```
>> A=rand(100);
>> B=sqrt(A);
```

In generale, si richiede la conoscenza delle funzioni *built–in* già vettorizzate e una buona comprensione del tipo di calcolo che si vuole svolgere.
Non esistono regole generali, va studiato caso per caso.

Vettorizzazione — 2

Consideriamo la funzione *minDistance*:

```
function d = minDistance(x,y,z)
nPoints = length(x);
d = zeros(nPoints,1);
for k=1:nPoints
        d(k) = sqrt(x(k)^2+y(k)^2+z(k)^2);
end
d=min(d);
```

Un esempio di utilizzo è il seguente:

```
>> x=rand(10000000,1); y=rand(10000000,1); z=rand(10000000,1);
>> tic;d=minDistance(x,y,z);toc
Elapsed time is 1.462213 seconds
```

Vettorizzazione — 3

Per vettorizzare questa funzione dobbiamo semplicemente rimpiazzare il *for loop* con un'operazione vettoriale. Il nuovo codice è:

```
function d = minDistance(x,y,z)
d = sqrt(x.^2+y.^2+z.^2);
d=min(d);
```

Si noti l'operatore · che agisce sull'intero vettore nel calcolo della funzione distanza. Il nuovo codice è solo leggermente più veloce:

```
>> x=rand(10000000,1); y=rand(10000000,1); z=rand(10000000,1);
>> tic;d=minDistance(x,y,z);toc
Elapsed time is 1.249956 seconds
```

ma dobbiamo attenderci vantaggi ben maggiori a seconda della dimensione dei dati.

Logica Vettorizzata

Spesso un calcolo realistico è composto oltre che da una parte computazionale da una parte dove sono realizzate operazioni condizionali (*logica*). Anche questo tipo di operazioni è vettorizzato in MATLAB mediante l'uso di funzioni specializzate come il *find*.
Ad esempio l'istruzione complessa:

```
>> i=find(isnan(x) |isinf(x));x(i)=[];
```

rimuove da un *dataset* chiamato *x* i valori *NaN* ed *Inf*.
Il codice risulta estremamente ottimizzato nonchè compatto. Sarebbe difficile ottenere entrambi gli obiettivi con un linguaggio come il C.

Integrare codice in MATLAB

- Una possibile alternativa all'ottimizzazione dello script è l'integrazione del proprio codice C o Fortran all'interno dell'ambiente MATLAB. A questo scopo la strada più semplice risulta scrivere un file di tipo *MEX* (Matlab EXecutable).
- Un file *MEX* si compone di una parte di calcolo (scritta nel linguaggio preferito C, Fortran, C++) e un *gateway* che interfaccia la parte di calcolo con MATLAB.
- Lo script *mex* si preoccupa di compilare il sorgente di tipo *MEX* (C o Fortan + chiamate alle *API*) e creare uno *shared object* dinamico che può essere richiamato all'interno di MATLAB.
- L'estensione del file *MEX* è dipendente dall'architettura e va quindi creata una copia per ogni piattaforma di calcolo che si vuole utilizzare.

Integrare codice in MATLAB — 2

Vediamo un esempio di file di tipo *MEX* Fortran che trasforma una matrice *piena* in una *sparsa*.

Integrare codice in MATLAB — 3

Dando il comando:

```
>> mex fulltosparse.f
```

creiamo il file fullotosparse.mexglx (Linux) che possiamo utilizzare da dentro MATLAB:

```
>> full = eye(5);
>> spar = fulltosparse(full)
spar =
   (1,1)         1
   (2,2)         1
   (3,3)         1
   (4,4)         1
   (5,5)         1
```

Ottimizzazione di codici Perl

- Perl è un linguaggio incredibilmente flessibile e semplice da utilizzare ma proprio per questo può soffrire di una certa perdita di prestazioni.
- Il motto di Perl *TMTOWTDI* (There's More Than One Way To Do It) comincia a vacillare se si debbono sviluppare applicazioni Perl su grande scala in cui la velocità di esecuzione è un requisito fondamentale.
- Singole istruzioni che danno lo stesso risultato possono avere differenze in termini di prestazioni macroscopiche. Questo fatto chiaramente si ripercuote sulle prestazioni dell'intero codice Perl.

Benchmark dei programmi Perl

- Il Benchmarking di codici Perl si effettua utilizzando il modulo *Benchmark* disponibile nella distribuzione standard di Perl.
- Vediamo un semplice esempio (bench-string.pl) che confronta due metodi per concatenare stringhe.
  - Il benchmark mostra chiaramente come il primo approccio sia da preferire.
  - Il problema del secondo caso è che l'istruzione *push* su un *array* è particolarmente costosa.
  - Viceversa, Perl risulta ottimizzato per operazioni su stringhe come quella presente nel primo caso.

Altre ottimizzazioni

- Si guadagna in velocità utilizzando i built-in del linguaggio (spostare le operazioni dalla parte interpretata a quella “compilata”)
- Avere *array* o *hash* di grandi dimensioni come argomenti di funzioni può essere pericoloso se non si usano alias o riferimenti.
- Fare chiamate a funzione è costoso, fare attenzione all’interno dei loop.

Come importare codice C in Perl

È disponibile il modulo *Inline* che consente di scrivere routines Perl in altri linguaggio (ma non Fortran). Un esempio:

```
use Inline C;
print "9 + 16 = ", add(9, 16), "\n";
print "9 - 16 = ", subtract(9, 16), "\n";
__END__
__C__
int add(int x, int y) {
  return x + y;
}
int subtract(int x, int y) {
  return x - y;
}
```

Profiling di script Python

- Python contiene due moduli per il profiling di script: *profile* e *hotshot*. I risultati prodotti sono processati dall'utility *pstats*. Esempio:

```
10020002 function calls in 107.714 CPU seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
10010000  75.808    0.000   75.808    0.000 int.py:14(f1)
  10000   31.880    0.003  107.688    0.011 int.py:4(Trapezoidal)
      1    0.026    0.026  107.713  107.713 int.py:2(?)
      1    0.001    0.001  107.714  107.714 <string>:1(?)
```

- Chiaramente la funzione *f1* e *Trapezoidal* sono quelle più costose in termini di *elapsed time*.
- Il tempo di CPU cresce di un fattore 5 usando *profile*
- Il modulo *hotshot* è notevolmente più veloce essendo scritto in C.

Ottimizzazione di codice Python

- *inlining* di funzioni Python. Le chiamate a funzione sono molto costose in Python. Soprattutto per piccole funzioni realizzare l'*inlining* nel codice dà uno *speed–up* nelle prestazioni
- Usare costrutti come `map`, `filter`, `reduce`
- Le variabili locali vengono accedute più velocemente di quelle globali
- utilizzare il costrutto *xrange* invece di *range* soprattutto per *loops* corposi
- *callbacks* a Python da Fortran, C, C++ sono molto costose
- Il costrutto *if–else* è notevolmente più performante del *try–except*.
- non utilizzare *loops* espliciti ma espressioni vettorizzate (vedi Numerical Python)

Numerical Python

- Il codice Python:

```
for i in range(len(x)):
    y[i] = sin(x[i])
```

  gira 20 volte più lento dell'equivalente C o Fortran (per $n \simeq 10^6$)

- Poichè tali operazioni sono comuni nel calcolo scientifico è stato sviluppato un pacchetto chiamato *Numerical Python*. Allora se x è un array in *NumPy* è possibile vettorizzare:

```
x = sin(x)
```

Altri tool

scipy Lo sviluppo di NumPy è stato interrotto e il progetto è confluito in *scipy*: oltre agli array e relative funzioni vettorizzate *scipy* fornisce supporto per la visualizzazione, decomposizione, integrazione numerica, processamento di segnali e immagini, algoritmi genetici, etc. . .

F2PY Consente di creare interfacce Python a funzioni Fortran e C.

Pyrex Un dialetto di Python che consente di mescolare tipi di dato Python e C e compila in estensioni C per Python.

Swig è uno strumento per importare codice C/C++ in svariati linguaggi di scripting e non. (e.g. Perl, Python, Tcl, Lisp, Java, C#, etc. . .)

Conclusioni

Quando non utilizzare un linguaggio di scripting

- l'applicazione implementa algoritmi complicati e presenta strutture dati dove può essere necessario un *tuning* molto fine per avere buone prestazioni
- l'applicativo maneggia grandi set di dati dove la gestione della memoria può essere un fattore critico
- i cambiamenti nel codice non avvengono troppo di frequente
- lo sviluppo dell'applicativo è affidato ad un team di persone e perciò è importante dichiarare staticamente le variabili ai fini di un miglior *debugging*

Quando utilizzare un linguaggio di scripting

- bisogna connettere insieme applicazioni già scritte
- l'applicazione include una interfaccia grafica
- l'applicazione deve fare notevole uso di processamento di testi
- si prevede che il *design* dell'applicativo cambi spesso
- le parti più pesanti in termini di CPU sono dislocate in punti ben determinati e si possono facilmente migrare a Fortran, C, C++.
- l'applicazione utilizza pesantemente strutture dati (eterogenee, nested) quali array ed Hash con gestione della memoria dinamica
- l'applicativo comunica con Web servers
- l'applicativo deve poter girare su Unix (Linux), Windows e Macintosh senza alcuna modifica

Bibliografia

- Matlab home page `http://www.mathworks.it/`
- Optimizing Matlab `http://www.mathworks.com/matlabcentral/files/5685/Writing\%20Fast\%20MATLAB\%20Code.pdf`
- GUI in Matlab, `http://zen.ece.ohiou.edu/~zhou/ComputerVision/GUIMatlab.htm`
- Matlab MEX, `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/ch5_fo12.html`
- Perl home page `http://www.perl.com/`
- Optimizing Perl, `http://www.developertutorials.com/print/153.html`
- L. Wall, T. Christiansen e J. Orwant, *Programming Perl*, O'Reilly
- S. Srinivasan, *Advanced Perl Programming*, O'Reilly.
- Python home page `http://www.python.org`

- Guido van Rossum, An Optimization Anecdote, `http://www.python.org/essays/list2str.html`
- H. P. Langtangen, *Python Scripting for Computational Science*, Springer
- Scientific Python Home Page, `http://www.scipy.org`
- SWIG, `http://www.swig.org/`