Julia (autore: Vittorio Albertoni)

Premessa

Il 14 febbraio 2012, al Massachusetts Institute of Technology, Jeff Bezanson, Stefan Karpinski, Viral Shah e Alan Edelman hanno reso disponibile un nuovo linguaggio di programmazione, da loro definito utile per il calcolo scientifico, il machine learning, il data mining, l'algebra lineare, il calcolo parallelo e il calcolo distribuito.

Si coronava così il sogno dei quattro volonterosi che da tre anni si erano posti il dichiarato obiettivo di superare i compromessi tra Matlab, Lisp, Python, Ruby, Perl, Mathematica, R e C, radunando in un solo linguaggio quanto di meglio è presente in quei linguaggi per il calcolo.

Il nome Julia è stato scelto senza particolari motivi: semplicemente perché da Karpinski è stato considerato un bel nome e un amico di Bezanson l'aveva suggerito.

Julia presenta i vantaggi di essere un linguaggio relativamente facile da apprendere e da usare e di eseguire calcoli, anche complicati e coinvolgenti masse enormi di dati, con una velocità non riscontrabile altrove nei così detti linguaggi interpretati, come Java, Python, Lisp, ecc., e pari a quella dei linguaggi compilati, come il C, tuttavia molto più difficile da apprendere e da utilizzare. Uno degli slogan di cui ama fregiarsi Julia è «Walks like Python. Runs like C».

La eccezionale velocità di elaborazione di questo linguaggio deriva dal fatto che Julia non lavora, come avviene negli altri casi di linguaggi interpretati, eseguendo le istruzioni dello script una per volta ma compila tutto lo script in linguaggio macchina e poi lo esegue. Da qui un riscontrabile piccolo ritardo nella partenza dell'esecuzione con un vistoso recupero nell'esecuzione una volta che è partita: ciò rende poco percepibile la velocità di Julia nell'esecuzione di script semplici, come quelli che affronteremo in questo manualetto, ma la velocità di Julia si manifesta nell'esecuzione di script impegnativi specialmente quando si è in presenza di grandi masse di dati da elaborare.

Queste caratteristiche fanno di Julia un linguaggio che promette di insidiare la primazia di Python per la data science e il big data mining.

Dal momento del rilascio Julia è diventato open source e ciò ha contribuito agli arricchimenti del linguaggio che si sono accumulati in pochi anni ma i suoi progettisti originari, con altri due soci, nel luglio del 2015 hanno fondato la società Julia Computing per "sviluppare prodotti che rendano Julia facile da usare, facile da implementare e facile da ridimensionare". Oltre a mantenere i repository open source Julia su GitHub, Julia Computing offre prodotti commerciali, incluso JuliaPro, disponibile sia in versione gratuita senza assistenza sia in versione a pagamento con assistenza professionale di primissimo livello.

In questo manualetto mi propongo di illustrare il funzionamento di base di Julia, conducendo il lettore alla realizzazione di script elementari da dilettanti.

Navigando in rete si possono trovare ottimi testi in lingua inglese per approfondimenti da professionisti, primo fra tutti il manuale ufficiale che troviamo, in formato PDF, all'indirizzo *https://julialang.org/*, nella pagina DOCUMENTATION.

Unico testo con un minimo di organicità su Julia in lingua italiana penso sia quello rintracciabile su *https://riptutorial.com/it/julia-lang* nel file `julia-language-it.pdf`, ma non è più aggiornato dalla superata versione di julia 0.5 del settembre 2016.

Indice

1 Installazione **3**

2 Come funziona **4**

3 Tipi **5**
- 3.1 Tipi base 5
  - 3.1.1 Numeri 5
  - 3.1.2 Caratteri 6
  - 3.1.3 Stringhe 6
  - 3.1.4 Valori booleani 6
- 3.2 Tipi contenitori 6
  - 3.2.1 Tupla 6
  - 3.2.2 Dizionario 6
  - 3.2.3 Array 7

4 Variabili **7**

5 Operatori **8**
- 5.1 Operatori aritmetici 8
- 5.2 Operatori di confronto 9
- 5.3 Operatori logici 9

6 Interattività con l'utente **9**
- 6.1 Output 9
- 6.2 Input 10

7 Strutture di controllo **10**
- 7.1 Esecuzione condizionale 10
- 7.2 Ripetizione 11

8 Semplici programmi **12**

9 Funzioni **15**
- 9.1 Funzioni preconfezionate di base 16
  - 9.1.1 Funzioni aritmetiche 16
  - 9.1.2 Funzioni trigonometriche 17
  - 9.1.3 Funzioni relative ad array numerici 17

10 Dot syntax e Funzioni anonime **17**

11 Librerie standard **18**
- 11.1 Printf 18
- 11.2 LinearAlgebra 18
- 11.3 Statistics 19

12 Packages **19**

13 Orientamento agli oggetti **20**

14 Lavorare con file **21**

15 Esercizi conclusivi **22**

1 Installazione

Julia si trova all'indirizzo *https://julialang.org/*.

Nel momento in cui scrivo (febbraio 2021), nella pagina DOWNLOAD è disponibile la versione 1.5.3, rilasciata il 9 novembre 2020, per Linux, Windows e Mac OS.

Nella stessa pagina troviamo il link per accedere a Julia Computing, che ci propone una versione gratuita, senza assistenza, di JuliaPro, che è la versione enterprise di Julia, equipaggiata con tutti i Packages disponibili e con la ricca IDE denominata Juno.

Per i lettori dilettanti cui mi rivolgo, e non solo, basta e avanza la versione normale.

Per un orientamento sulla scelta faccio presente che l'installazione di Julia occuperà circa 360 MB del nostro disco mentre per l'installazione di JuliaPro dovremo disporre di circa 5 GB liberi.

Da tenere anche presente che tutti i Packages inclusi in JuliaPro possono essere installati individualmente anche in Julia, se e quando servono, nei modi che vedremo.

Sistema Linux

La versione per Linux che scarichiamo dal sito è racchiusa in un archivio `.tar` e l'installazione avviene semplicemente scompattando questo archivio in una qualsiasi directory (consigliabile la directory di sistema `/opt`): Julia sarà così contenuto nella directory `julia-1.5.3` e l'eseguibile `julia` nella sottodirectory `bin`.

Perché Julia possa essere lanciato in qualsiasi posizione ci troviamo a lavorare occorre inserire nella variabile di sistema PATH il percorso all'eseguibile.
A tal fine individuiamo il file `.bashrc` sul nostro computer (nelle ultime versioni di Ubuntu si tratta del file `bash.bashrc` nella directory `/etc`) e, aprendo un editor di testo con poteri di root, nell'ipotesi che abbiamo scompattato Julia in `/opt`, aggiungiamo la seguente riga
`export PATH=$PATH:/opt/julia-1.5.3/bin`.

Se ci accontentiamo della versione di Julia che ci passa il repository della nostra distro, che difficilmente corrisponderà all'ultima rilasciata, possiamo installare Julia ricorrendo all'installatore di programmi. In Ubuntu, collegati a internet, con il comando a terminale
`sudo apt install julia`.

Se ci accontentiamo della versione classic 1.0.4 rilasciata nel maggio 2019 possiamo infine ricorrere a snap. In Ubuntu, collegati a internet, con il comando a terminale
`sudo snap install julia --classic`.

In questi due casi non abbiamo nemmeno il problema del PATH in quanto tutto sarà fatto con l'installazione.

Si tenga conto che, per ciò che vedremo in questo manualetto, basta una versione da 1.0 in poi.

Sistema Windows

Per Windows scarichiamo un installer che provvede a tutto, offrendoci anche un launcher con icona per lanciare Julia.

Se vogliamo poter lanciare Julia da terminale per eseguire uno script memorizzato trovandoci posizionati in una qualsiasi directory diversa da quella che contiene l'eseguibile `julia.exe`, dobbiamo inserire nella variabile di sistema PATH il percorso a questo eseguibile.
A tal fine accediamo alle variabili d'ambiente attraverso le impostazioni di sistema avanzate e aggiungiamo alla variabile PATH il percorso.

Sistema Mac OS X

Per Mac scarichiamo un file `.dmg` che va trattato come prevede il sistema operativo.

Eventuali problemi di PATH si risolvono, come visto per Linux, aggiungendo l'istruzione `export PATH=$PATH:`<percorso> nel file `.bash_profile` che si trova nella directory `Home`.

2 Come funziona

Julia è un linguaggio interpretato e interattivo.

Interpretato significa che il codice che scriviamo viene eseguito direttamente da uno script.

Interattivo significa che possiamo anche vedere eseguite le nostre istruzioni intanto che le scriviamo.

Quello che abbiamo installato secondo quanto indicato nel precedente Capitolo è l'interprete e viene avviato con il comando a terminale `julia`.[1].

Se ci proponiamo di realizzare il nostro primo programma Ciao mondo possiamo sperimentarlo in due modi.

Possiamo scrivere in un file di testo, con un qualsiasi editor di testo, questa istruzione

```
println("Ciao mondo")
```

e salvare il file con estensione `.jl`: ciaomondo.jl.

Successivamente, e tutte le volte che vogliamo, posizionandoci nella directory dove abbiamo salvato il file e scrivendo a terminale

```
julia ciaomondo.jl
```

vedremo eseguito il nostro programma (meglio chiamato script) che produrrà la voluta scritta.

L'altro modo è quello, interattivo, di avviare l'interprete scrivendo a terminale il comando

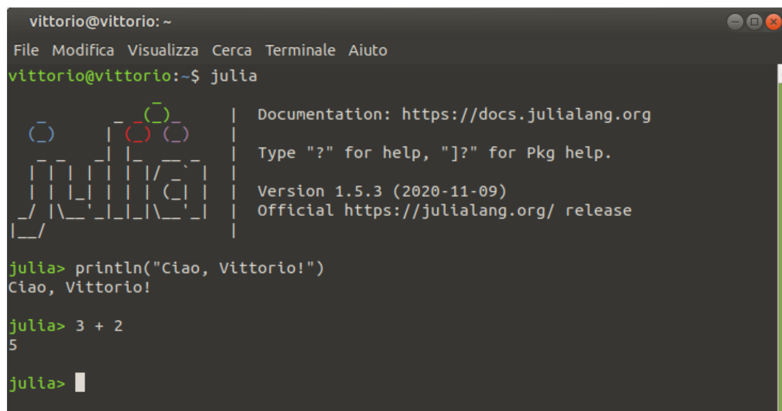
```
julia
```

col che il terminale si trasforma in terminale Julia, la shell Julia (chiamata anche REPL, per Read-Evaluate-Print-Loop), una conchiglia all'interno della quale si usa il linguaggio Julia, e vi possiamo scrivere l'istruzione

```
println("Ciao mondo")
```

vedendola immediatamente eseguita alla pressione del tasto INVIO.

La shell di Julia, nel mio terminale Linux Ubuntu, si presenta così



La modalità interattiva è molto utile per vedere in diretta l'effetto delle nostre istruzioni.

Nell'illustrazione vediamo a terminale l'istruzione julia che apre la shell e mostra il prompt `julia>`.

Scriviamo in corrispondenza al prompt un'istruzione in linguaggio Julia e premendo Invio ne vediamo immediatamente l'effetto: sono esemplificate una istruzione per un saluto e una istruzione per sommare due numeri.

Quando l'effetto dell'istruzione è un risultato numerico, l'ultimo risultato ottenuto viene memorizzato in una variabile chiamata `ans` (che sta per answer, risposta). Nel caso dell'illustrazione, se, al punto in cui siamo, scriviamo l'istruzione `ans * 6` otteniamo il risultato `30`.

Se non ci interessa vedere il risultato dell'istruzione, la terminiamo con un punto e virgola (`;`). In tal modo non vedremo il risultato, che verrà tuttavia memorizzato nella variabile `ans`. Così, se scriviamo `3 * 2;` non vedremo il risultato `6` premendo INVIO, ma, se immediatamente dopo scriviamo `ans + 2`, vedremo il risultato `8`.

[1] Il terminale è una finestrella nella quale possiamo scrivere istruzioni con cui interagire con il computer. In Linux e Mac OS X si chiama proprio Terminale e troviamo modo di avviarlo dai menu a disposizione. In Windows si chiama Prompt dei comandi e si avvia con il comando `cmd` o `command`.

La shell di Julia può anche essere utilizzata per accedere al volo alla descrizione di una istruzione del linguaggio.
Per fare questo, in corrispondenza del prompt `julia>` scriviamo `?` e il prompt si trasforma nel pront `help?>`. Se, in corrispondenza di questo nuovo prompt scriviamo un'istruzione del linguaggio, ad esempio `println`, ne otterremo la descrizione ed esemplificazioni sul suo utilizzo.

La shell di Julia può anche momentaneamente tornare ad essere terminale di sistema digitando il carattere punto e virgola in corrispondenza al suo prompt.

Se premiamo INVIO nella shell di Julia prima di aver completato un'istruzione che Julia riconosca possiamo completare l'istruzione nella riga successiva e solo dopo che l'istruzione sarà completa e riconosciuta, alla pressione di INVIO, verrà eseguita.

Se vogliamo far eseguire a Julia, attraverso la shell, più istruzioni insieme dobbiamo elencarle tra la parola chiave `begin` e la parola chiave `end` oppure scriverle, separate da punto e virgola, tra parentesi tonde.
Ad esempio, se scriviamo al prompt

```
begin
print("3 volte 2 fa ")
3*2
end
```

alla pressione di INVIO dopo aver scritto `end` abbiamo il risultato
`3 volte 2 fa 6.`

Come se avessimo scritto `(print("3 volte 2 fa "); 3 * 2).`

Possiamo scoprire altre finezze sulla shell Julia andando nella Parte III, Standard Library, del manuale di Julia, al Capitolo 81, The Julia REPL.

3 Tipi

Julia è un linguaggio a tipizzazione dinamica e non richiede che il tipo di dato da elaborare sia preventivamente indicato dall'utente: a seconda della sintassi con cui è espresso il dato da elaborare Julia assegna al dato stesso il tipo.

I tipi base di Julia sono i soliti che troviamo nei linguaggi di programmazione (numeri, caratteri, stringhe).

3.1 Tipi base

3.1.1 Numeri

Julia tratta numeri interi, numeri in virgola mobile (dotati del separatore decimale `.`) e numeri complessi.

Per i numeri interi (`Int`) esiste un'ampia scelta ricollegabile alla dimensione: si va da `Int8` (intero con segno a 8 bit compreso -2^7 e $2^7 - 1$, cioè tra -128 a 127) a `Int128` (intero con segno a 128 bit compreso tra 0 e $2^{127} - 1$, che è un bel numero a 39 cifre). Nei sistemi operativi a 64 bit Julia assegna automaticamente al dato numerico intero (senza il punto decimale) il tipo `Int64` (intero di 19 cifre).

Per i numeri in virgola mobile (`Float`), quelli indicati con il punto separatore decimale, abbiamo due tipi: `Float32` per la precisione singola (8 cifre significative) e `Float64` per la precisione doppia (16 cifre significative). Nei sistemi operativi a 64 bit Julia assegna automaticamente al dato numerico con punto di separazione decimale il tipo Float64.

Tutte queste limitazioni dimensionali sono finalizzate alla velocità di elaborazione. Se dobbiamo lavorare con grandi numeri possiamo rinunciare alla velocità e lavorare in precisione arbitraria ottenendo numeri grandi quanto ce lo consente l'hardware su cui lavoriamo (come avviene normalmente se usiamo Python). A questo scopo basta che un solo operando venga scritto come `BigInt(`<numero>`)` o come `BigFloat(`<numero>`)`, anche semplicemente `big(`<numero>`)`.

Nei numeri complessi (`Complex`) parte reale e parte immaginaria, dotate di segno, sono scritte di seguito e la parte immaginaria è contrassegnata dal suffisso `im`.

Esempi:
`728` è un numero intero,
`7.5` è un numero in virgola mobile,
`-1+0.78im` è un numero complesso.

3.1.2 Caratteri

Il carattere (`Char`) è un carattere racchiuso tra apici semplici (`'`).

3.1.3 Stringhe

La stringa (`String`) è una sequenza di caratteri racchiusa tra apici doppi (`"`).

La stringa è immutabile, cioè non è possibile aggiungervi altri caratteri o toglierne.

La posizione del carattere è indicizzata partendo da 1 (non da 0 come avviene quasi sempre in informatica). La lunghezza della stringa viene indicata dalla funzione `length(`<stringa>`)`.

Esempi:
`"Vittorio"` è una stringa,
`"Vittorio"[5]` è il quinto carattere della stringa "Vittorio", cioè o.

3.1.4 Valori booleani

I valori booleani (`Bool`) sono `true` (vero) e `false` (falso).

3.2 Tipi contenitori

Sono insiemi variamente organizzati di elementi di un tipo base.

L'elemento può essere indicato letteralmente o attraverso il richiamo di una variabile che lo contiene. Il suo numero viene ottenuto con la funzione `length()`.

3.2.1 Tupla

La tupla (`Tuple`) è una sequenza di elementi eterogenei, racchiusi tra parentesi tonde e separati da una virgola (,).

La tupla è immutabile, cioè non è possibile aggiungervi altri elementi o toglierne.

La posizione dei sui elementi è indicizzata partendo da 1.

Esempio:
`(1, 24, "giuseppe", 'p', A)` è una tupla, composta da due interi, una stringa, un carattere e dal contenuto della esistente variabile A.

3.2.2 Dizionario

Il dizionario (`Dict`) è una sequenza di elementi eterogenei contraddistinti da una chiave accoppiata a ciascun elemento.

Viene creato inserendo chiavi e elementi tra due parentesi tonde precedute dalla parola chiave `Dict`.

Chiave e relativo elemento sono separati dal simbolo => e gli accoppiamenti sono separati da virgola (,).

Esempio:
`Dict('a' => 35, 3 => "pippo", 5 => 78)` è un dizionario e `'a'`, `3` e `5` sono le chiavi.
`Dict('a' => 35, 3 => "pippo", 5 => 78)[3]` individua l'elemento corrispondente alla chiave `3`, cioè la stringa `"pippo"`.

3.2.3 Array

L'array (Array) è una collezione ordinata di elementi racchiusi tra parentesi quadre ([e]) e disponibili su più righe e colonne. Gli elementi di ogni riga si indicano separati da uno spazio e inserendo un punto e virgola si passa a una riga successiva. Il numero di elementi per ogni riga, che corrisponde al numero di colonne, deve sempre essere uguale.

Un array può essere composto da una sola riga (indicando gli elementi separati da spazio): in tal caso si parla anche di matrice monodimensionale; oppure può essere composto di una sola colonna (indicando gli elementi separati dal punto e virgola): in tal caso si parla di vettore.

L'array con più righe e più colonne è una matrice pluridimensionale.

La posizione degli elementi è indicizzata partendo da 1. Per gli array su una sola riga o su una sola colonna l'indice indica la posizione nella riga, partendo da sinistra, o nella colonna, partendo dall'alto. Per gli array su più righe la posizione è individuata da un indicatore composto dall'indice di riga e, separato da una virgola, dall'indice di colonna.

Esempi:
['a' 2 "pippo"] è un array su una sola riga,
['a' 2 "pippo"][3] individua il suo terzo elemento "pippo",
[5; "Luigi"; true] è un array su una sola colonna,
[5; "Luigi"; true][2] individua il suo secondo elemento "Luigi",
[1 'g' 12; "pippo" 24 'a'] è un array di due righe e tre colonne,
[1 'g' 12; "pippo" 24 'a'][2,3] individua il terzo elemento sulla seconda riga 'a'.

Gli array esclusivamente composti di elementi numerici sono di fondamentale importanza per l'algebra lineare.

Per chi conosce il linguaggio Python, che non contempla l'array se non attraverso il modulo aggiuntivo Numpy e, nella versione di base, contempla il tipo List, sottolineo come, in Julia, dove invece abbiamo l'array ma non la lista, l'array a una dimensione possa essere utilizzato come lista, cioè un tipo contenitore di un numero variabile di elementi eterogenei.

Un array può essere inizializzato con il comando [ ] e gli elementi possono essere inseriti con push!(<array>, <elemento>) per creare un array colonna (vettore).

Inserendo più vettori colonna separati da spazio tra [e] costruiamo una matrice. In uno script occorre preventivamente inizializzare la matrice con [ ] e inserire la prima colonna con la funzione vcat[<matrice> <colonna>].

* * *

Esiste la funzione typeof(), attraverso la quale possiamo verificare quale tipo Julia assegna alla cosa che scriviamo.

Per esempio, se scriviamo nella shell di Julia typeof(122), viene ritornato Int64, a dimostrazione del fatto che Julia ha capito che abbiamo scritto un numero intero e automaticamente lo tratta a 64 bit.

4 Variabili

Le variabili sono delle locazioni di memoria, delle scatole, alle quali diamo un nome, destinate a contenere valori di un certo tipo.

In Julia le variabili si creano nel momento in cui servono, senza bisogno, come avviene in molti linguaggi di programmazione, di dichiararle prima.

La tipizzazione della variabile avviene in maniera dinamica automaticamente al momento dell'assegnazione del valore, in quanto Julia, come abbiamo visto nel precedente Capitolo, riconosce il tipo da come scriviamo il valore stesso.

L'assegnazione del valore viene fatta con l'operatore = e la sintassi
<nome_variabile> = <valore>.
nome = "Vittorio" crea la variabile nome e le assegna il valore di tipo String Vittorio
raggio = 6.5 crea la variabile raggio e le assegna il valore di tipo Float64 6,5

A = [15 22 8.5] crea l'array (vettore riga) A assegnandogli i valori 15 22 8.5
l = [] crea la variabile l contenente un array/lista vuoto
push!(l, 15) inserisce nell'array/lista contenuto nella variabile l l'elemento numerico 15.

Il valore può essere espresso in modo letterale, come negli esempi, o in forma di espressione matematica, più o meno coinvolgendo altre variabili.

Può accadere di dover modificare il tipo di una variabile, per esempio per poter applicare ad essa metodi che non sarebbero propri del tipo assegnato dinamicamente al momento della sua creazione.

I casi più frequenti sono quelli di dover convertire in stringa un valore numerico e viceversa.

Per convertire un numero in stringa usiamo la funzione
string()
passando come argomento tra le parentesi il numero da convertire o il nome della variabile che lo contiene.

Per convertire una stringa in numero usiamo la funzione
parse(<tipo>, <cosa>)
dove <tipo> è il tipo numerico che scegliamo: se indichiamo Int Julia intende Int64 oppure indichiamo noi il tipo voluto; per il tipo Float dobbiamo indicare noi Float32 o Float64;
<cosa> è la stringa da convertire o il nome della variabile che la contiene.

Le variabili create nello script sono variabili globali. Le variabili create in un blocco di istruzioni all'interno dello script (ne riparleremo quando parleremo di cicli di controllo e di funzioni) sono variabili locali. In un blocco di istruzioni interno allo script si usano normalmente variabili locali; se vogliamo utilizzare una variabile globale dobbiamo richiamarla con il suo nome preceduto dalla parola chiave global.

Per le variabili globali non è possibile indicare il tipo di valore che sono destinate a contenere. Esso viene esclusivamente determinato dalla natura del dato inserito.

Per la variabili locali possiamo determinare il tipo nei modi che vedremo parlando delle funzioni.

5 Operatori

Gli operatori collegano tra loro operandi in espressioni che forniscono un risultato.

5.1 Operatori aritmetici

In ordine di precedenza di esecuzione sono i seguenti:
^ per l'elevamento a potenza,
// per la creazione di una frazione,
* per la moltiplicazione,
/ per la divisione,
% per il modulo (resto della divisione intera),
+ per la somma,
- per la sottrazione.

L'operatore per la creazione di frazioni penso sia esclusivo del linguaggio Julia. Inserito tra numeratore e denominatore crea una frazione che si può assoggettare a calcolo, come fosse un numero, e il risultato viene espresso a sua volta come frazione.
Esempi:
3//2 + 1 dà 5//2
3//2 + 1//2 dà 2//1
(5//4)^2 dà 25//16

La trasformazione di una frazione nel corrispondente numero decimale si può fare con
Float64(<frazione>)
Esempio:
Float64(25//16) dà 1.5625

Gli operatori `+` e `-` possono essere utilizzati per l'addizione o la sottrazione tra matrici numeriche della stessa dimensione.

Gli operatori `*` `/` e `^` possono essere utilizzati per il prodotto, la divisione e l'elevamento a potenza di matrici, rammentando che prodotto e divisione si possono fare se le colonne della prima matrice sono tante quante le righe della seconda e l'elevamento a potenza si può fare solo per le matrici quadrate.

Per ottenere prodotto, divisione ed elevamento a potenza membro a membro occorre far precedere all'operatore il simbolo punto (`.`).

Ovviamente `*` e `/` si utilizzano anche per moltiplicazioni e divisioni tra matrici e scalari, rammentando che, nelle divisioni, lo scalare può essere soltanto il secondo membro dell'operazione.

5.2 Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano. Sono i seguenti:
== uguale,
`!`= non uguale,
< minore,
<= minore o uguale,
> maggiore,
>= maggiore o uguale.
Gli operandi assoggettati al confronto devono avere lo stesso tipo.

5.3 Operatori logici

Forniscono come risultato un valore booleano e sono i seguenti:
`&&` per l'AND logico,
`||` per l'OR logico.

6 Interattività con l'utente

L'interfacciamento con l'utente avviene normalmente attraverso il terminale.

Esiste la possibilità di creare GUI utilizzando le librerie Gtk o il framework Tk ma non affronto queste cose in questo manualetto per dilettanti.

6.1 Output

Le funzioni per l'output sono `print` e `println` con la sintassi
`print`(<cosa_scrivere>)
`println`(<cosa_scrivere>)
dove <cosa_scrivere> può essere indicato con una stringa, una espressione matematica o il nome di una variabile che contiene il valore che vogliamo scrivere, anche combinati tra loro separati con una virgola (`,`).

La differenza tra le due è che la prima scrive senza andare a capo mentre la seconda scrive e va a capo.

Esempi:
`println("ciao")` scrive `ciao` e va a capo
`println(5)` scrive `5` e va a capo,
`print(3*7.5)` scrive `22.5` e non va a capo,
`println("3 per 8 fa ", 3*8)` scrive `3 per 8 fa 24` e va a capo,
data la variabile `x` contenente il valore `16`,
`print(x)` scrive `16` e non va a capo,
`print("la variabile x vale ", x)` scrive `la variabile x vale 16` e non va a capo.

Per formattare l'output dobbiamo ricorrere ad un modulo che vedremo.

6.2 Input

La funzione per l'input è `readline` con la seguente sintassi
`readline()`

L'esecuzione del programma resta sospesa fino a quando l'utente non ha inserito da tastiera il dato richiesto.

Eventuali istruzioni su che cosa l'utente debba inserire vanno impartite con un precedente messaggio scritto con `println()`.

Il dato inserito viene letto come stringa.

Se ciò che serve è un numero dobbiamo ricorrere alla funzione `parse()`, che abbiamo visto alla fine del Capitolo 4, per convertire immediatamente la stringa letta in numero.

Esempi:
`x = readline()` inserisce quanto digitato nella variabile `x` come stringa,
`y = parse(Float64, readline())` inserisce un numero, decimale a 64 bit, nella variabile `y`,
`println(2^(parse(Int, readline())))` se inseriamo `3` scrive `8`.

7 Strutture di controllo

Come ogni altro linguaggio di programmazione, Julia ha dei comandi per condizionare l'esecuzione di certe istruzioni al verificarsi di determinate condizioni oppure per la ripetizione dell'esecuzione di una o più istruzioni.

7.1 Esecuzione condizionale

if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if <condizione>
<istruzioni>
end
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite la o le istruzioni indicate prima della parola chiave `end`, altrimenti si passa oltre.

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

```
if <condizione>
<istruzioni>
else
<istruzioni>
end
```

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco, prima della parola chiave `else`, altrimenti vengono eseguite quelle contenute nel secondo blocco dopo `else` (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `elseif` con questa sintassi

```
if <condizione>
<istruzioni>
elseif <condizione>
<istruzioni>
elseif <condizione>
<istruzioni>
else
<istruzioni>
end
```

7.2 Ripetizione

for

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni per un numero definito di volte.

La sintassi per l'uso di questa istruzione in Julia è molto simile a quella che ritroviamo in Python:

```
for <range>
<istruzioni>
end
```

La situazione più semplice è quella che utilizza una sorta di contatore, una variabile creata al volo tradizionalmente indicata con `i`, che assuma un certo numero di valori:

```
for i = 1:5
println("Ciao")
end
```

scrive `Ciao` cinque volte.

Numero delle iterazioni ed elementi su cui eseguirle possono essere determinati utilizzando un qualsiasi oggetto che sia una sequenza.
Per esempio:

```
for x in "pippo"
println(x)
end
```

elenca su cinque righe le lettere che compongono la stringa `"pippo"`

```
for x in "pippo"
println("Ciao")
end
```

scrive `Ciao` cinque volte, quante sono le lettere che compongono la stringa `"pippo"`

while

Si usa per ripetere istruzioni fino a quando si verifica una certa condizione.

La sintassi è:

```
while <condizione>
<istruzioni>
end
```

Se la condizione è espressa attraverso l'uso di un contatore otteniamo gli stessi risultati che otteniamo con l'istruzione `for` vista prima

```
i = 1
while i <= 5
print "Ciao"
i = i + 1
end
```

scrive cinque volte la parola `Ciao`.

L'istruzione `while`, combinata con `break`, si presta ad interrompere un ciclo al verificarsi di un certo evento, come l'inserimento di un certa stringa da tastiera:
con
```
while true
x = readline()
if x == "fine"
break
end
end
```
viene richiesto un input fino a quando non si scrive la parola `fine`. Attenzione ai due `end`: il primo chiude il blocco di istruzioni `if` e il secondo chiude il blocco di istruzioni `while`.

8 Semplici programmi

Quanto visto finora ci consente di creare i nostri primi semplici script.

Cominciamo da questo, banalissimo, che chiede il nome all'utente e gli rivolge un saluto personalizzato:

```
println("Come ti chiami?")
nome = readline()
println("Ciao, ", nome, "!")
```

Quest'altro calcola la circonferenza e l'area di un cerchio chiedendone il raggio all'utente:

```
println("Digita il raggio di un cerchio")
r = parse(Float64, readline())
println("Per un cerchio di raggio ", r)
println("La circonferenza è ", r * 2 * pi)
println("e l'area è ", r^2 * pi)
```

L'esempio mi dà modo di ricordare che la costante matematica pi greco, richiamabile con `pi`, è l'unica costante matematica richiamabile senza aggiungere librerie a Julia di base.

Ora qualche cosa di più impegnativo, con uno script che chiede l'inserimento di una doppia serie di dati e calcola l'indice della correlazione esistente tra loro, determina l'equazione della retta di regressione lineare ed il relativo coefficiente di determinazione. Come dire che un po' di machine learning si può fare anche artigianalmente, senza ricorrere a chissà quali librerie.
L'indice di correlazione di Pearson si calcola con la formula

$$r = \frac{\sum_i[(x_i-\bar{x})(y_i-\bar{y})]}{\sqrt{\sum_i(x_i-\bar{x})^2}\sqrt{\sum_i(y_i-\bar{y})^2}}$$

Il coefficiente angolare della retta di regressione nella forma

$$y = a + bx$$

si calcola con la formula

$$b = \frac{\sum_i[(x_i-\bar{x})(y_i-\bar{y})]}{\sum_i(x_i-\bar{x})^2}$$

e l'intercetta si calcola con la formula

$$a = \bar{y} - b\bar{x}$$

Infine il coefficiente di determinazione è il quadrato dell'indice di correlazione.
Rammento che $\bar{x}$ e $\bar{y}$ sono rispettivamente la media delle x e la media delle y.

Lo script è riportato nella pagina seguente.

Ho utilizzato le indentazioni unicamente per rendere più visibili i blocchi di istruzioni e migliorare la lettura dello script. Le indentazioni non sono richieste da Julia.

```
sx = 0.0
sy = 0.0
n = 0
lx = Float64[]
ly = Float64[]
spscxy = 0.0
sqscx = 0.0
sqscy = 0.0
println("Inserisci i dati (f per finire)")
println("prima la x poi la y")
while true
    xx = readline()
    yy = readline()
    if xx == "f"
        break
        end
    x = parse(Float64, xx)
    push!(lx, x)
    y = parse(Float64, yy)
    push!(ly, y)
    global sx = sx + x
    global sy = sy + y
    global n = n + 1
    end
mx = sx / n
my = sy / n
for i in 1:n
    global sqscx = sqscx + (lx[i] - mx)^2
    end
for i in 1:n
    global sqscy = sqscy + (ly[i] - my)^2
    end
for i in 1:n
    global spscxy = spscxy + (lx[i] - mx) * (ly[i] - my)
    end
r = spscxy / (sqscx^(1/2) * sqscy^(1/2))
b = spscxy/sqscx
a = my - b * mx
println("dati:")
for i in 1:n
    println(lx[i], " ", ly[i])
    end
println()
println("Coefficiente di correlazione:")
println(r)
println()
println("Retta di regressione:")
println("y = ", a , " + ", b, "x")
println()
println("Coefficiente di determinazione:")
println(r^2)
```

Innanzi tutto abbiamo l'inizializzazione a valore zero delle variabili globali che dobbiamo avere a disposizione per far lavorare le formule (con `0.0` otteniamo una variabile destinata a contenere un tipo Float64 e con `0` otteniamo una variabile destinata a contenere un tipo Int64):
`sx` e `sy` sono destinate a contenere la somma delle `x` e la somma delle `y`,
`n` è destinata a contenere il numero delle coppie di dati `x` e `y`,
`lx` e `ly` predispongono gli array destinati a contenere i valori `x` e `y`,
`spscxy` conterrà la somma dei prodotti degli scarti delle `x` e delle `y` dalle loro medie
(il numeratore delle formule per calcolare `r` e `b`),

sqscx e sqscy conterranno le somme dei quadrati degli scarti delle x e delle y dalle loro medie, (per i denominatori delle formule per calcolare r e b).

Il successivo blocco while è destinato all'immissione dei dati, prevedendo il blocco all'inserimento della lettera f al posto dei due dati richiesti. I dati man mano inseriti vengono immessi nei rispettivi array, vengono sommati e contati.

Segue il calcolo delle medie.

Abbiamo poi tre cicli for in cui calcoliamo le sommatorie dei quadrati degli scarti e dei prodotti degli scarti.

Infine lo sviluppo delle formule e la presentazione dei risultati.

Nello sviluppo della formula per il calcolo della r, visto che ancora non conosciamo come si calcola la radice quadrata, ho utilizzato l'elevamento alla potenza 1/2.

Chiudo la serie di questi primi esercizi con questo, nel quale costruiamo una matrice lavorando con la tastiera.

```
println("Creiamo una matrice.")
print("Quante colonne?  ")
c = parse(Int, readline())
print("Quante righe?  ")
r = parse(Int, readline())
M = []
colonna = []
println()
println("Colonna 1")
for ii = 1:r
    println("Elemento ", ii)
    push!(colonna, parse(Float64, readline()))
    end
M = vcat(M, colonna)
for i = 2:c
    col = []
    println()
    println("Colonna ", i)
    for ii = 1:r
        println("Elemento ", ii)
        push!(col, parse(Float64, readline()))
    end
    global M = [M col]
end
println()
println(M)
```

Cominciamo chiedendo all'utente di quante colonne e di quante righe debba essere la matrice.

Inizializziamo poi una variabile globale, chiamata M, destinata a contenere la matrice.

Poi costruiamo la prima colonna: inizializzata la variabile colonna destinata a contenerla, avviamo un ciclo per chiedere all'utente gli elementi nel numero voluto.

Con la funzione vcat inseriamo questa prima colonna nella matrice, conferendo così alla matrice una struttura del numero di righe voluto.

Avviamo poi il ciclo per le successive colonne, con annidato il ciclo per inserirvi gli elementi. All'interno del ciclo le colonne che creiamo le chiamiamo con un nome diverso (col) per non confonderle con la variabile globale (colonna) che avevamo creato per la prima colonna. Sempre all'interno del ciclo aggiungiamo ciascuna colonna alla matrice M.

Usciti dal ciclo stampiamo la matrice.

* * *

I programmi che abbiamo visto si salvano in file con estensione `.jl` e si lanciano nel terminale con il comando `julia` seguito dal nome del file.

Chi lavora in Linux può fare in modo che essi siano lanciabili semplicemente con il loro nome o cliccando sul loro nome, una volta resi eseguibili con `chmod` o agendo sulle Proprietà/Permessi nel gestore dei file. Per questo basta inserire nella prima riga, preceduto dai caratteri `#!`, il percorso e il nome dell'eseguibile `julia`.
Nel caso che Julia sia nella directory `opt` dovremmo scrivere

```
#!/opt/julia-1.5.3/bin/julia
```

Alla fine dello script, per fare in modo che il terminale non si chiuda senza darci modo di vedere il risultato delle elaborazioni, dovremo inserire l'istruzione `readline()`, magari preceduta dal messaggio «Premi Invio per uscire»

9 Funzioni

La funzione è una sezione del programma in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che la funzione è chiamata.

Per semplificare la scrittura di un certo programma, potrebbe essere utile raggruppare alcune istruzioni all'interno del programma stesso, creando funzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per definire una funzione è

```
function <nome_funzione> (<nome_parametro>, <nome_parametro>,...)
<istruzioni>
end
```

dove <nome_funzione> è il nome che intendiamo dare alla funzione, ed è il nome che useremo per richiamarla, <nome_parametro> è il nome del o dei parametri (dati) da inserire quando la richiamiamo.
Le <istruzioni> sono quelle relative alle elaborazioni da effettuare sui parametri per ottenere il risultato.

La funzione restituisce il valore dell'ultima espressione inserita nelle istruzioni. Con l'istruzione `return` prima di `end` possiamo stabilire noi che cosa debba restituire la funzione.

Possiamo stabilire il tipo dei parametri attraverso l'operatore `::` inserito tra il <nome_parametro> e il tipo voluto.

Possiamo stabilire il tipo del valore restituito facendo seguire all'elenco dei parametri l'operatore `::` a sua volta seguito dal tipo voluto.

La sintassi per chiamare una funzione ed eseguirla è

```
<nome_funzione> (<parametro>, <parametro>,...)
```

Esempi:
definita la seguente funzione

```
function saluta(nome)
print("Ciao ", nome)
end
```

con l'istruzione
`saluta("Giuseppe")` scriviamo `Ciao Giuseppe`.
definita la funzione

```
function area_triangolo(base::Float32, altezza::Float32)::Float64
base * altezza / 2
end
```

con l'istruzione
`area_triangolo(12, 6)` otteniamo il risultato `36.0` espresso come Float64 e gli interi 12 e 6 che abbiamo inserito come parametri vengono letti come Float32.

Non sapendo ancora che esiste una funzione preconfezionata per calcolare il fattoriale di un numero scriviamola noi:

```
function fattoriale(n :: Int) :: Int128
if n == 0 return 1
else return n * fattoriale(n-1)
end
end
```

Si impone che il parametro da passare alla funzione sia un numero intero (se si passa un decimale il programma dà errore ed è giusto che sia così, in quanto il fattoriale esiste solo per i numeri interi) e che il risultato sia un intero a 128 bit, in modo da poter arrivare a calcolare il fattoriale di 33.

Dal momento che questa è una limitazione inaccettabile se vogliamo, per esempio, utilizzare la funzione per il calcolo combinatorio, conviene togliere le tipizzazioni e, quando si richiama la funzione, indicare il parametro n come `BigInt(n)` o semplicemente come `big(n)` in modo da portarci in precisione arbitraria e poter calcolare anche il fattoriale, per esempio, di 1345.

La funzione diventa

```
function fattoriale(n)
if n == 0 return 1
else return n * fattoriale(n-1)
end
end
```

e, richiamata in questo modo

```
fattoriale(BigInt(1345))
```

ci mostrerà un bel numero di 3626 cifre.

9.1 Funzioni preconfezionate di base

Julia ci offre tutta una serie di funzioni utili per tante cose senza che dobbiamo invocare alcuna libreria per poterle utilizzare.

Alcune le conosciamo già in quanto le ho utilizzate nei vari esempi.

Pensiamo alle funzioni `print()` e `println()` per l'output, alla funzione `readline()` per l'input e alla funzione `BigInt()` che abbiamo appena usato. Alle funzioni che abbiamo utilizzato per lavorare con array e matrici, come `push!()` e `vcat()`.

Moltissime altre ce ne sono e le troviamo descritte nella Parte II, Base, del manuale.

Qui elenco quelle di più ricorrente utilità in campo matematico. L'elenco completo si trova nel Paragrafo 45.2 del manuale.

9.1.1 Funzioni aritmetiche

Consentono la manipolazione di dati numerici.

`abs(n)` restituisce il valore assoluto del valore numerico n,
`exp(n)` restituisce la potenza ennesima del numero e,
`round(n)` restituisce, sotto forma di numero intero, la parte intera di n arrotondata,
`trunc(n)` restituisce, sotto forma di numero intero, la parte intera di n senza arrotondamenti,
`log(n)` restituisce il logaritmo naturale di n,
`log10(n)` restituisce il logaritmo decimale di n,
`log(b,n)` restituisce il logaritmo in base b di n,
`pi` restituisce il valore della costante π,
`sqrt(n)` restituisce la radice quadrata di n,
`factorial(n)` restituisce il fattoriale di n,
`hypot(x,y)` restituisce l'ipotenusa con cateti x e y,
`lcm(n, n, n, ....)` restituisce il mimimo comune multiplo dei numeri indicati.
Per la potenza ennesima di x possiamo usare la combinata `exp(log(x)*n)`.
Per la radice ennesima di x possiamo usare la combinata `exp(log(x)/n)`.

9.1.2 Funzioni trigonometriche

Esprimendo n in radianti abbiamo le funzioni dirette `sin(n)`, `cos(n)` e `tan(n)`.
Esprimendo n in gradi abbiamo le funzioni dirette `sind(n)`, `cosd(n)` e `tand(n)`.
Le funzioni inverse asin(n), acos(n) e atan(n) restituiscono l'angolo in radianti.
Le funzioni inverse asind(n), acosd(n) e atand(n) restituiscono l'angolo in gradi.
`deg2rad(n)` converte gradi in radianti.
`rad2deg(n)` converte radianti in gradi.

9.1.3 Funzioni relative ad array numerici

length(<array>) restituisce il numero degli elementi,
minimum(<array>) restituisce il valore minimo contenuto,
maximum(<array>) restituisce il valore massimo contenuto.

10 Dot syntax e Funzioni anonime

Il linguaggio Julia possiede una particolarità, che non si trova in altri linguaggi di programmazione, riguardante i calcoli che coinvolgono array: attraverso l'uso di un punto `(.)` ben collocato insieme a operatori o richiami di funzione si ottiene quella che possiamo chiamare vettorizzazione di un'operazione matematica.

Nel Capitolo 5 abbiamo visto come si possano utilizzare normalmente, con quali effetti e con quali limiti, i normali operatori aritmetici tra array o tra array e scalari.

Ci rimane qualche problema con l'elevamento a potenza di un array, possibile con il semplice operatore ^, a patto che la base sia una matrice quadrata ed ottenendo come risultato l'elevamento a potenza matriciale.

Già allora abbiamo visto che per ottenere l'elevamento a potenza membro a membro occorre far precedere all'operatore ^ un punto `(.)` ed abbiamo così anticipato la tecnica della dot syntax.

In questo modo non esiste più l'esigenza che la base della potenza sia una matrice quadrata, ma essa può essere una matrice qualsiasi o un array monodimensionale.

Così, in presenza dell'array A, con
`A .^ 3`
otteniamo un nuovo array i cui elementi sono il cubo degli elementi dell'array A.

Allo stesso modo possiamo sommare uno scalare a un array o sottrarre uno scalare da un array:
A .+ <scalare> oppure A .- <scalare>

La cosa si fa ancor più interessante se estendiamo il discorso alle funzioni.

Tutte le funzioni che abbiamo visto nel precedente Capitolo 9, e quante ne possiamo costruire noi, se prevedono come argomento un numero, possiamo utilizzarle passando come argomento un array, di qualsiasi tipo e dimensione, ottenendo un nuovo array i cui elementi sono il risultato dell'applicazione della funzione a ciascun elemento dell'array di partenza.

Così, in presenza dell'array A, con
<nome_funzione>.(A)
otteniamo un nuovo array i cui elementi sono il risultato dell'applicazione della funzione a ciascun elemento dell'array di partenza.

Altra particolarità di Julia è quella delle funzioni anonime, utili per semplificare la generazione di un array attraverso calcoli applicati agli elementi di un altro array.

La sintassi per generare una funzione anonima con argomento x è la seguente
x -> <espressione>
Così con
`x -> x^(1/2)`
generiamo una funzione anonima che calcola la radice quadrata di `x`.

L'uso della funzione anonima torna utile ogniqualvolta dobbiamo passarla come argomento ad un'altra funzione.

Esiste, per esempio, la funzione `map`, destinata a generare un insieme di dati applicando una certa funzione ad un altro insieme di dati. La sua sintassi è

```
map(<funzione>, <dati>)
```

Per generare un array formato dalle radici quadrate dell'insieme di dati contenuto nell'array A basta fare

```
map(x -> x^(1/2), A)
```

11 Librerie standard

Una libreria è una raccolta di funzioni.

Le funzioni che abbiamo visto finora fanno parte dalla grande libreria di base e possiamo richiamarle direttamente.

Una volta installato Julia abbiamo però a disposizione tantissime altre funzioni che sono raccolte in altre librerie.

Alcune di queste librerie sono disponibili senza installare alcunché di aggiuntivo a Julia. Esse sono già installate con Julia e, per servircene, dobbiamo semplicemente richiamarle con la sintassi

`using` <nome_libreria>.

Queste sono le librerie raggruppate nella Stantard Library e le troviamo descritte nella Parte III del Manuale di Julia.

Vi sono poi moltissime altre funzioni che fanno parte di librerie che non vengono installate con Julia (salvo ricorrere a Julia-Pro): sono i così detti Packages.

Per utilizzare queste funzioni dobbiamo preventivamente installare il package di cui fanno parte.

In questo capitolo presento sommariamente alcune librerie che fanno parte della Standard Library.

11.1 Printf

Contiene la funzione utile per formattare l'output.

Per utilizzarla occorre l'istruzione

```
using Printf
```

Abbiamo così a disposizione la direttiva di formattazione

```
"%.<cifre_decimali>f"
```

utilizzabile con la funzione `@printf` per stabilire le cifre decimali da scrivere arrotondando l'ultima.

Per esempio:

`print(7.168 * 4.68)` scrive `33.54624`

`@printf "%.2f" 7.168*4.68` scrive `33.55`

`@printf "%.5f" pi` scrive `3.14159`

11.2 LinearAlgebra

Contiene funzioni per l'algebra lineare.

Abbiamo visto che già con la dotazione base di Julia abbiamo a disposizione funzioni per l'aritmetica delle matrici (addizione, sottrazione, prodotto, divisione).

In questa libreria troviamo funzioni per l'algebra lineare e, per utilizzarla, occorre l'istruzione

```
using LinearAlgebra
```

Abbiamo così a disposizione, tra le altre, le funzioni

`det(M)` per calcolare il determinante della matrice M,

`inv(M)` per calcolare la matrice inversa di M.

11.3 Statistics

Contiene funzioni statistiche.

Per utilizzarla occorre l'istruzione
`using Statistics`

Tra le altre, abbiamo a disposizione le funzioni
`mean(`<array>`)` per calcolare la media della serie contenuta in un array,
`std(`<array>`)` per calcolare la deviazione standard campionaria della serie contenuta in un array,
`var(`<array>`)` per calcolare la varianza campionaria della serie contenuta in un array,
`cor(`<vettore_x>, <vettore_y>`)` per calcolare l'indice di correlazione di Pearson tra una serie di x e una serie di corrispondenti y contenute in due vettori (array verticali).

12 Packages

I packages sono librerie destinate ad arricchire le funzionalità di base del linguaggio e sono disponibili nel repository open source di Julia in GitHub.

Una panoramica generale dei pacchetti del mondo Julia la troviamo all'indirizzo
https://julialang.org/packages/
Attraverso il link *JuliaHub* possiamo ricercare, conoscendone il nome, un pacchetto per poi acquisirne la documentazione.
Attraverso il link *Julia Packages* possiamo consultare l'elenco dei pacchetti esistenti divisi per categoria.

All'indirizzo *https://github.com/trending/julia* abbiamo anche l'elenco aggiornato dei pacchetti che vanno per la maggiore (trending).

Per utilizzare un package dobbiamo innanzi tutto installarlo

L'installazione avviene utilizzando la shell di Julia in modalità Pkg.

Per entrare in questa modalità inseriamo al prompt di Julia una parentesi quadra chiusa `]`. Il prompt si trasformerà così da `julia>` a `pkg>`.

Da qui, con il comando ? otteniamo l'elenco di tutti i comandi previsti per la gestione dei pacchetti con una sommaria descrizione del loro scopo e con il comando ? seguito dal nome di un comando otteniamo la descrizione completa della sintassi per utilizzare il comando stesso.

I comandi di più ricorrente utilizzo sono:
`add` <nome_pacchetto> per installare un pacchetto,
`rm` <nome_pacchetto> per disinstallare un pacchetto,
`st` per vedere l'elenco dei pacchetti installati.

La disinstallazione con `rm` disattiva il pacchetto ma non rimuove i file che costituiscono il pacchetto dal disco e li tiene in cache al fine di velocizzare una eventuale reinstallazione.

Esiste il comando `gc` (che sta per garbage collector) che ci consente di liberare spazio di memoria occupato da rimasugli di pacchetti disinstallati e non più usati per un certo periodo (mi pare di aver capito attorno ai 30 giorni).

E' importante tener presente queste cose in quanto i pacchetti di Julia tendono ed essere alquanto pesanti in termini di occupazione di spazio su disco.

Per uscire dalla modalità Pkg usiamo il tasto BACKSPACE o la combinazione CTRL - C.

Per poter usare un pacchetto dobbiamo inserire nello script, prima di usarlo, il comando
`using` <nome_pacchetto>

La prima volta che si usa il pacchetto, e solo in questa occasione, occorre attendere qualche tempo affinché si compia la precompilazione.

Tra i tanti pacchetti cito innanzi tutto quelli più legati alla data science e alla matematica:
Flux per il machine learning,
Plots e **Winston** per grafici normali,
Gadfly e **Makie** per grafici super,
DifferentialEquations per la soluzione di equazioni differenziali,

Gen per la statistica avanzata.

Abbiamo anche **Tk** per la costruzione di GUI.

Per lavorare con i database possiamo ricorrere ai packages **MySQL** e **SQLite**.

Per lavorare con dataset su file in formato `.csv` abbiamo il package **CSV.**

Interessante la possibilità di utilizzare in Julia funzioni del linguaggio Python attraverso il package **PyCall**, importando il modulo Python che le contiene.
Per default questo pacchetto, alla sua installazione, si collega all'ecosistema Python3 presente sul computer e, ovviamente, i moduli richiamati devono essere installati.

Dopo la dichiarazione `using PayCall`, l'importazione del modulo che interessa avviene creando un oggetto che contiene le funzioni del modulo stesso con
<nome_oggetto> `= pyimport(`<stringa_nome_modulo>`)`
dopo di che ogni funzione è richiamabile con
<nome_oggetto>`.`<nome_funzione>
come se lavorassimo in Python.

Se, per esempio, vogliamo calcolare l'integrale definito tra 0 e 2 della funzione x^2 utilizzando il sotto-modulo `integrate` di `scipy`, basta scrivere nella shell di Julia o in uno script da memorizzare quanto segue

```
using PyCall
i = pyimport("scipy.integrate")
i.quad(x->x^2, 0, 2)
```

e con questo misto di linguaggio Julia e Python otteniamo il risultato in questa tupla

```
(2.666666666666667, 2.960594732333751e-14)
```

il cui primo valore è quello dell'integrale e il secondo è l'errore di approssimazione del calcolo.

Il package PyCall dà accesso a tutto l'ecosistema Python che abbiamo installato sul computer.

Per certi moduli Python esistono specifici packages Julia di interfacciamento che rendono ancor più agevole l'integrazione tra i linguaggi. Penso, per esempio ai packages **PyPlot** e **SymPy**.
Di notevole importanza quest'ultimo, grazie al quale possiamo arricchire Julia della potenzialità del calcolo simbolico che altrimenti non avrebbe[2].

Teniamo comunque presente che quanto abbiamo descritto in questo manualetto può tranquillamente essere praticato su un computer non particolarmente dotato di risorse in termini di velocità del processore e di RAM.

Se utilizziamo pacchetti aggiuntivi al sistema di base qui trattato teniamo presente che, specialmente i pacchetti di grafica dell'ecosistema Julia, sono particolarmente esosi di risorse e, con computer datati o dotati di risorse limitate, potremmo avere difficoltà, soprattutto in termini di tempo di esecuzione dei nostri script.

13 Orientamento agli oggetti

Non si può dire che Julia sia un linguaggio che agevola la programmazione per oggetti e non consiglio a un dilettante di provarci.

Tuttavia in Julia è tutto un oggetto, anche se, abituati ad altri linguaggi, non ce ne accorgiamo, visto che Julia utilizza una sintassi tutta sua per lavorare con gli oggetti.

Generalmente, quando si richiama la funzione membro di un oggetto, si usa la sintassi
<nome_oggetto>`.`<funzione_con_relativi_parametri>.

In Julia la sintassi è
<nome_funzione>(<nome_oggetto>, <parametri_della_funzione>).

Prendiamo, per esempio, la funzione con cui si aggiunge un elemento a un vettore.
In Julia la utilizziamo così
`push!(`<vettore>`, `<elemento>`)`

[2] Sul calcolo simbolico segnalo il mio articolo «Software libero per il calcolo simbolico» del settembre 2019 sul mio blog all'indirizzzo *www.vittal.it*, con allegato il manualetto «sympy.pdf», buona guida introduttiva a SymPy.

e in Python, dove avremmo una lista al posto del vettore, la utilizziamo così, come succederebbe in Java e C++
<lista>`.append(`<elemento>`)`

14 Lavorare con file

Julia ci offre tante modalità per lavorare con i file, alcune semplificanti ma pericolosissime per la salute dei file stessi se non usate con la dovuta attenzione.

Qui rammento la più sicura, molto simile a quella che ritroviamo in altri linguaggi, basata sulla funzione `open()` con la quale creiamo un oggetto `file` dotato di funzioni per scriverne, aggiornarne o leggerne il contenuto.

La sintassi è la seguente
`f = open(<nome_file>, <modalità>)`
dove `f` è un qualsiasi nome che diamo all'oggetto `file`,
`<nome_file>` è una stringa che indica il percorso al file,
`<modalità>` è una stringa che può assumere i seguenti valori:
`"w"` se vogliamo scrivere sul file, creandolo se non c'è o sostituendolo,
`"a"` se vogliamo aggiungere elementi ad un file esistente senza sostituirlo,
`"r"` se vogliamo leggere il file.
Le più importanti funzioni membro del nostro oggetto file sono
`write(<oggetto_file>, <stringa>)` per scrivere una stringa nel file (per andare a capo occorre terminare la stringa con `\n`),
`read(<oggetto_file>, String)` per leggere tutto il file come unica stringa,
`readline(<oggetto_file>)` per leggere una riga del file (una volta letta una riga ci si posiziona sulla riga successiva e occorre ripetere il comando per leggerla),
`readlines(<oggetto_file>)` per leggere tutto il file e inserirne il contenuto in un array i cui elementi sono costituiti da una riga del file.

Per rendere operativo ciò che abbiamo fatto sul file e chiuderlo posizionandosi sul primo elemento occorre utilizzare il metodo
`close(<oggetto_file>)`

Esempi:
con `f = open("/home/vittorio/Documenti/prova", "w")`
creo il file `prova` nella posizione indicata dal percorso,
con `write(f, "Pippo\n")` scrivo la sua prima riga con il nome Pippo,
con `close(f)` chiudo il file, confermando quanto scritto in esso,
con `f = open("/home/vittorio/Documenti/prova", "a")`
apro il file `prova` per aggiungere altri elementi
con `write(f, "Pluto\n", "Paperino\n")`
aggiungo altre due righe,
con `close(f)` chiudo il file, confermando quanto aggiunto,
con `f = open("/home/vittorio/Documenti/prova", "r")`
apro il file `prova` per leggerne il contenuto
con `readline(f)` leggo la prima riga e, se ripeto il comando subito dopo, leggo la seconda e così via
con `close(f)` chiudo il file e riposiziono tutto.
Se riapro il file in lettura,
con `readlines(f)` ottengo l'array
`"Pippo"`
`"Pluto"`
`"Paperino"`
con `read(f, String)` otterrei la stringa
`"Pippo\nPluto\nPaperino\n"`

Il metodo `write` può scrivere una sola stringa per volta, per cui se vogliamo scrivere più stringhe dobbiamo separarle con una virgola e se vogliamo scrivere numeri dobbiamo lavorare di conversione di tipo.

Per esempio, con il comando

```
write(f, "3 moltiplicato 2 fa ", string(3 * 2), "\n")
```

scriveremmo nel nostro file la riga

```
3 moltiplicato 2 fa 6
```

Per elaborare file di dati su più righe e più colonne, usualmente in formato `.csv`, dobbiamo ricorrere al package CSV.

15 Esercizi conclusivi

Per esemplificare un po' tutto ciò che abbiamo visto anche negli ultimi Capitoli propongo un paio di script.

Il primo serve per risolvere un sistema di equazioni lineari di qualsiasi dimensione.

```
using LinearAlgebra
println("Risoluzione di un sistema di equazioni lineari.")
print("Quante equazioni?  ")
n = parse(Int, readline())
M = Float64[]
colonna = Float64[]
println()
println("Coefficienti delle X1")
for ii = 1:n
println("equazione ", ii)
push!(colonna, parse(Float64, readline()))
end
M = vcat(M, colonna)
for i = 2:n
    col = Float64[]
    println()
    println("Coefficienti delle X", i)
    for ii = 1:n
        println("equazione ", ii)
        push!(col, parse(Float64, readline()))
    end
    global M = [M col]
end
println()
println("Termini noti:")
N = []
for i = 1:n
    push!(N, parse(Float64, readline()))
end
x = inv(M) * N
println()
println("Soluzioni:")
for i = 1:n
    print("X",i, " = ")
    println(x[i])
end
```

La risoluzione del sistema è ottenuta attraverso la nota formula

$$X = M^{-1}N$$

dove X è il vettore delle soluzioni, M è la matrice dei coefficienti delle incognite (M^{-1} ne è l'inversa) e N è il vettore dei termini noti.
La costruzione della matrice, semplificata dal fatto che ora dobbiamo costruire una matrice quadrata, avviene come abbiamo visto in un esercizio nel Capitolo 8.
La novità è rappresentata dall'applicazione della formula per ottenere la soluzione e dalle cinque righe finali per scriverle.
Per disporre della funzione di inversione della matrice abbiamo importato la libreria standard LinearAlgebra (nelle ultime versioni di Julia sarebbe comunque disponibile).

Con quest'altro script possiamo risolvere un'equazione, calcolando le radici di equazioni anche di grado elevato.

Se abbiamo installato il package PyCall lo script potrebbe essere il seguente

```
using PyCall
s = pyimport("sympy")
println("Ricerca delle radici di un'equazione quasiasi")
println("Inserisci la parte sinistra di una equazione uguagliata a zero")
println("(esprimere utilizzando gli operatori ^ * / + -)")
espressione = readline()
x = s.symbols("x")
e = s.sympify(espressione)
soluzioni = s.solve(e, x)
for i = 1:length(soluzioni)
    println("soluzione ", i, ":  ", s.N(soluzioni[i]))
end
```

Se abbiamo installato solo SymPy o se vogliamo comunque utilizzare quello, lo script potrebbe essere il seguente

```
using SymPy
println("Ricerca delle radici di un'equazione quasiasi")
println("Inserisci la parte sinistra di una equazione uguagliata a zero")
println("(esprimere utilizzando gli operatori ^ * / + -)")
espressione = readline()
x = symbols("x")
e = sympify(espressione)
soluzioni = solve(e, x)
for i = 1:length(soluzioni)
    println("soluzione ", i, ":  ", N(soluzioni[i]))
end
```

In questo caso siamo in pieno calcolo simbolico e ricorriamo, in un modo o in un altro, alla potenza di Python.
Le istruzioni evidenziate in rosso sono nel linguaggio di SymPy e, come si vede, convivono con istruzioni in linguaggio Julia per arrivare a risultati strabilianti con uno script così piccolo.
Lavorando in questo modo, tuttavia, si nota un po' meno la decantata velocità di Julia.