Grafica per Julia (autore: Vittorio Albertoni)

Premessa

Questo manualetto è da considerarsi complemento di quello intitolato «julia» allegato all'articolo «Un nuovo linguaggio per la data science» pubblicato nel febbraio 2021 sul mio blog all'indirizzo *www.vittal.it*.

In quella sede ho illustrato le basi del linguaggio ed ho mostrato con quale facilità Julia ci consenta di fare calcoli, anche non banali, interagendo con il computer attraverso la tastiera e lo schermo.

A cosa serve la grafica di cui ci occuperemo qui?

Sostanzialmente a due cose: la prima importante, la seconda, almeno a mio giudizio, un po' meno.

La prima riguarda l'arricchimento che la grafica può fornire alle tecniche di rappresentazione di dati.

La seconda riguarda la possibilità di costruire interfacce grafiche per rapportarsi con il computer: non più solo tastiera ma anche mouse e finestre contenenti istruzioni su come agire, scelte da effettuare, ecc.

I padri di Julia hanno creato Julia Computing, poi divenuta JuliaHub, che, oltre ad offrire prodotti commerciali a pagamento con assistenza di altissimo livello, mantiene i repository open source di Julia su GitHub. Inoltre Julia è stato concepito per interfacciarsi il più possibile con pacchetti open source prodotti con altri linguaggi di programmazione, Python in primis, ma non solo.

Esiste pertanto una miriade di pacchetti che possiamo utilizzare per espandere la potenza di Julia e moltissimi di questi riguardano la grafica.

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).

Consultando queste fonti possiamo avere un'idea di cosa stia diventando l'ecosistema di Julia: nel momento in cui scrivo (giugno 2026) nella sola categoria dei pacchetti di grafica cui siamo qui interessati se ne contano ben 140.

La consultazione ci mostra anche come questo mondo sia tutt'altro che stabilizzato, sia ancora in pieno fermento di arricchimento e come la strada per introdurvisi da parte di un dilettante sia alquanto impervia: la poca documentazione esistente, ovviamente scritta sempre in lingua inglese, è sovente per esperti addetti ai lavori e per tantissimi pacchetti la documentazione non esiste affatto.

Ciò premesso, mi accingo ad illustrare ciò che di più facile e documentato ho trovato per la grafica dei dati e per la grafica dell'interfaccia utente, in modo che anche un dilettante possa disporre di qualche strumento e sapere come utilizzarlo.

Rammento che per installare un pacchetto, dalla shell di Julia, inserendo il carattere della parentesi quadra chiusa (`]`) e premendo INVIO, passiamo alla shell dell'installatore dei pacchetti e, da questa shell, si installa il pacchetto con il comando `add` seguito dal nome del pacchetto stesso. Con il tasto BACKSPACE si torna alla shell di Julia.

Indice

I Grafica per la rappresentazione di dati **3**

1 Plots **3**
1.1 plot() 3
1.2 scatter() 6
1.3 bar() 6
1.4 pie() 7

2 Makie **7**

3 PyPlot **8**

II Grafica per interfaccia utente (GUI) **13**

Parte I
Grafica per la rappresentazione di dati

Tutti sappiamo quanto aiuti la grafica a rendere meglio leggibili e interpretabili i dati.

In Statistica linee, torte e canne d'organo rendono immediata la comprensione di trend, di ripartizioni e di raffronti come nessuna tabella può fare.

In Matematica i grafici di funzione rendono l'idea di sintesi del comportamento di una funzione matematica come nessun indicatore analitico può fare.

Julia ci mette a disposizione moltissimi pacchetti originali per fare queste cose e ci consente di utilizzare con estrema facilità gli strumenti aventi le stesse finalità che troviamo nell'ecosistema Python.

Se siamo votati a Julia in esclusiva e non abbiamo Python sul nostro sistema possiamo scegliere tra i moltissimi strumenti originali.

Se abbiamo sul nostro sistema anche Python arricchito degli strumenti per le ricerche scientifiche e la scienza dei dati possiamo creare dei mix molto efficienti tra i due linguaggi con il vantaggio di poter scegliere il meglio dell'uno e dell'altro.

1 Plots

Plots è il package originale dell'ecosistema Julia che ha sostituito il package Winston, rispetto al quale è molto più ricco.

E' diventata la libreria di visualizzazione più diffusa e versatile dell'ecosistema Julia.

Ha il pregio di creare grafici potendo utilizzare diversi motori grafici, senza dover modificare il codice.

I motori grafici che può utilizzare sono:
. `GR` quello di default, molto veloce, ideale per uso normale,
. `PlotlyJS` per grafici interattivi,
. `PythonPlot` basato su Matplotlib di Python,
. `PGFPlotsX` con qualità tipografica per LaTeX,
. `UnicodePlots` per grafici direttamente nel terminale.

Installando Plots si installa anche il motore GR.

Per utilizzare altri motori grafici occorre installare, oltre a Plots, i relativi package, che hanno il nome indicato nell'elenco appena visto.

Direi che per principianti basta e avanza che ci si accontenti del motore di default GR.

Anche perché per grafici interattivi è molto meglio il package di cui parlo nel prossimo Capitolo e per grafici basati su Matplotlib di Python ci dà meno problemi agire come vedremo nel prossimo Capitolo 3.

Per il resto si tratta di perfezionamenti da professionisti.

Per usare Plots occorre dichiararne l'intenzione all'inizio dello script e subito dopo dichiarare il motore grafico preferito, con questa sintassi:

```
using Plots
gr()
```

La seconda indicazione non è necessaria se vogliamo usare il motore GR ma è obbligatoria se vogliamo usare un altro motore.

All'indirizzo *https://docs.juliaplots.org/stable/* troviamo un ottimo manuale di Plots in lingua inglese.

Qui presento una piccola rassegna delle funzioni principali di questo potente strumento grafico.

1.1 plot()

La funzione `plot()`, per default, traccia un grafico a linea.

Gli argomenti necessari sono gli array contenenti i dati di base per il tracciamento.

In presenza di un solo array `y` dei dati per le ordinate, con `plot(y)` tracciamo una linea che congiunge i dati dell'array in corrispondenza delle ascisse 1, 2, 3, 4, ...

In presenza di due array `x` per le ascisse e `y` per le ordinate, con `plot(x,y)` tracciamo una linea che congiunge le corrispondenze tra le x e le y.

Tra le parentesi tonde, oltre agli argomenti, possiamo inserire attributi per il grafico. Ricordo qui solo gli attributi principali:

. attributi per titoli e etichette:

`title` per esprimere con una stringa il titolo del grafico,

`label` per etichettare il grafico con una stringa (con valore `false` l'etichetta scompare),

`xl` per esprimere con una stringa l'etichetta dell'asse delle ascisse,

`yl` per esprimere con una stringa l'etichetta dell'asse delle ordinate,

. attributi per modificare l'aspetto visivo della linea o del tipo di grafico:

`color` per esprimere il colore della linea con sintassi `:`<nome_colore> ,

`lw` per esprimere lo spessore della linea con un numero,

`ls` per esprimere lo stile della linea con sintassi `:solid`, `:dash`, `:dot`, `:dashdot`,

`st` per definire il tipo di grafico con possibile scelta tra i principali

- `:path` per la linea, che è il default,
- `:scatter` per i punti,
- `:bar` per un istogramma a barre.

. altri attributi:

`grid` per mostrare o nascondere la griglia di sfondo (`true` o `false`),

`size` per dimensionare il grafico in pixel come tupla (es. `size=(800, 600)`),

`bg` per esprimere un colore di sfondo del grafico con sintassi `:`<nome_colore>.

* * *

Con la funzione `plot!` possiamo realizzare la modifica incrementale di un grafico, cioè aggiungere un grafico ad uno già esistente.

* * *

Le funzioni plot e plot! si prestano egregiamente per rappresentare grafici di funzioni matematiche, accettando come argomenti la funzione da rappresentare e il campo di variazione.

Se la funzione è riconosciuta da Julia, basta scriverne il nome:

`plot(sin, -pi, pi)` traccia il grafico della funzione $y = \sin(x)$ nell'intervallo di ascisse -π: π

Se la funzione non è riconosciuta da Julia possiamo crearla noi come funzione anonima:

`plot(x -> x^2, -5, 5)` traccia il grafico della funzione $y = x^2$ nell'intervallo di ascisse -5:5.

Per definire sia l'intervallo delle ascisse sia quello delle ordinate da visualizzare occorre usare gli attributi `xlims` e `ylims`, uguagliandoli alla tupla contenente gli estremi degli intervalli.

Con qualche complicazione in più possiamo creare grafici tridimensionali per funzioni di due variabili, utilizzando il comando `surface` applicato agli array relativi ai valori delle x e delle y e alla funzione $f(x, y)$.

* * *

Per salvare un grafico prodotto con `plot()` possiamo assegnarlo a una variabile e poi salvarlo con la funzione

`savefig(`<nome_variabile>`,` <nome_grafico>`)`

dove <nome_grafico> è il nome da dare al grafico, completo di percorso, con estensione `.png`, `.pdf` o `.svg`, a seconda del formato voluto.

* * *

Possiamo produrre il grafico dalla REPL di Julia inserendo via via le istruzioni.

Se vogliamo salvare lo script in un file da lanciare quando serve con l'interprete Julia, dobbiamo avere l'avvertenza di inserire, alla fine dello script, le istruzioni

```
gui()
readline()
```

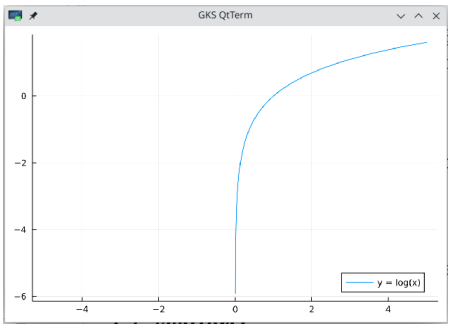
La prima per forzare la produzione del grafico, la seconda per farlo rimanere visibile fino a quando non si preme un tasto.

* * *

Con questo comando nella REPL

```
plot(log, -5, 5, label = "y = log(x)")
```

otteniamo



dove abbiamo l'asse delle ascisse visualizzato tra -5 e 5 e quello delle ordinate visualizzato a scelta del programma alla dimensione utile per la lettura del grafico, scartando automaticamente le zone prive di dati (zona superiore) o prive di dati significativi (zona inferiore con l'asintoto).

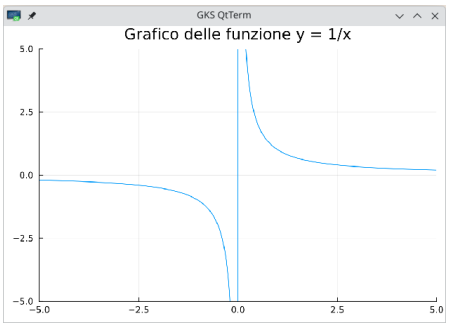
Se passiamo il comando

```
plot(x->1/x, -5, 5, title = "Grafico delle funzione y = 1/x")
```

otteniamo un grafico illeggibile e dobbiamo governare noi anche l'asse delle ordinate in questo modo

```
plot(x->1/x, xlims=(-5, 5),ylims=(-5, 5), title = "Grafico delle funzione y = 1/x", legend=false)
```

ottenendo



Con questo script tracciamo il grafico tridimensionale della funzione

$z = x^2 - y^2$

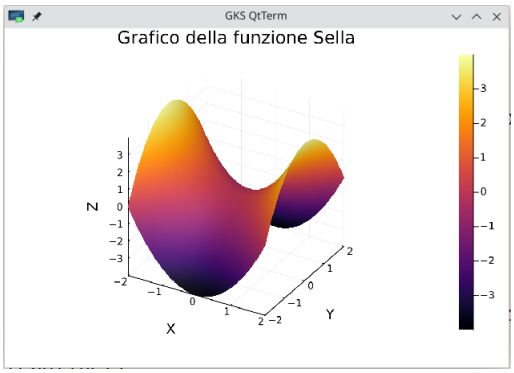
detta Funzione a sella (paraboloide iperbolico)

```
using Plots
f(x, y) = x^2 - y^2
x = range(-2, 2, length=30)
y = range(-2, 2, length=30)
surface(x, y, f, title="Grafico della funzione Sella",xl="X", yl="Y", zl="Z")
gui()
readline()
```

Il piccolo trucco sta nel ridefinire la funzione `z` come funzione di due variabili `f(x,y)`e definire gli intervalli per gli assi x e y.

Passando poi, insieme agli array x e y la funzione f a `surface`, Julia valuta la funzione su ogni combinazione di x e y.

Lanciando lo script con Julia otteniamo



1.2 scatter()

La funzione `scatter()` disegna sul piano i punti che rappresentano i dati, senza unirli con una linea.

Possiamo ottenere la stessa cosa con la funzione `plot()`, definendo opportunamente l'attributo `st`.

Gli attributi di abbellimento del grafico sono per lo più gli stessi della funzione `plot()`.

Notevole l'attributo `smooth`, dando al quale il valore `true` insieme al grafico dei punti viene tracciata la retta di regressione.

* * *

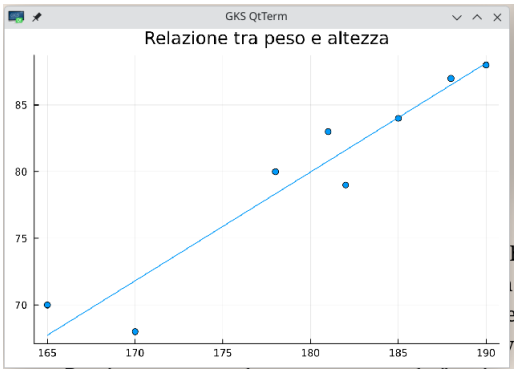
Dati gli array

```
x = [165,170,178,181,182,185,188,190]
y = [70,68,80,83,79,84,87,88]
```

che relazionano l'altezza (x) e il peso (y) di un piccolo campione di persone, con questo comando nella REPL

```
scatter(x, y, title = "Relazione tra peso e altezza", smooth=true, legend=false)
```

otteniamo



1.3 bar()

La funzione `bar()` fa apparire il valore dell'ordinata corrispondente ad una ascissa con una barra verticale.

Possiamo ottenere la stessa cosa con la funzione `plot()`, definendo opportunamente l'attributo `st`.

Gli attributi di abbellimento del grafico sono per lo più gli stessi della funzione `plot()`.

Il grafico a barre è utile per far corrispondere una serie di dati rappresentati in un array `y` a delle categorie classificatorie rappresentate in un array di stringhe `x`.

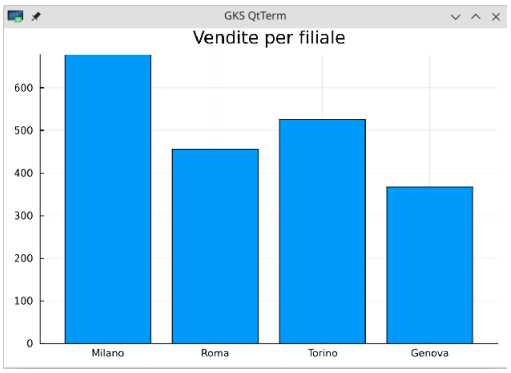
L'importo delle vendite delle nostre filiali di Milano, Roma, Torino e Genova è

Milano	678
Roma	456
Torino	525
Genova	367

Scrivendo nella REPL

```
x = ["Milano", "Roma", "Torino", "Genova"]
y = [678, 456, 525, 367]
bar(x, y, title = "Vendite per filiale", legend = false)
```

otteniamo



1.4 pie()

La funzione `pie()` produce una torta le cui fette, differenziate per colore, sono proporzionali all'entità dei valori delle ordinate.

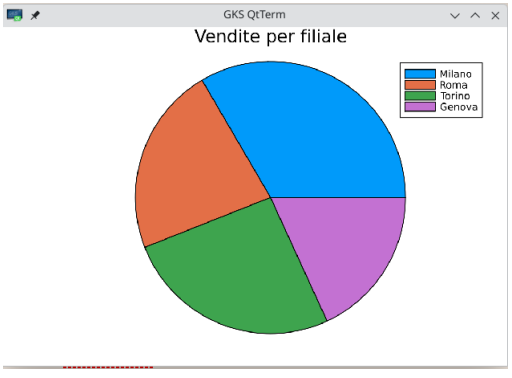
E' utile, come il grafico a barre, per far corrispondere una serie di dati rappresentati in un array `y` a delle categorie classificatorie rappresentate in un array di stringhe `x`.

* * *

Con le stesse array sulle vendite per filiale dell'esempio precedente, scrivendo nella REPL

```
pie(x, y, title = "Vendite per filiale")
```

otteniamo



2 Makie

Anche Makie è un package originale dell'ecosistema Julia, probabilmente il migliore e il più completo.

L'utilizzo del pacchetto è alquanto impegnativo, sia sul piano della difficoltà di apprendimento sia sul piano delle risorse: se non abbiamo particolari inclinazioni informatiche è meglio che non lo utilizziamo e se lo utilizziamo su computer non particolarmente dotati dobbiamo pazientare sui tempi di risposta.

All'indirizzo *https://makie.juliaplots.org/stable/* è comunque discretamente documentato in una facile lingua inglese ed ho ritenuto utile citarlo per chi voglia provare a cimentarsi con esso utilizzando questa documentazione, di una completezza difficilmente riscontrabile per altri pacchetti dell'ecosistema Julia.

Lo possiamo installare attraverso uno dei quattro backend packages che lo rendono disponibile: GLMakie.jl, CairoMakie.jl, WGLMakie.jl, RPRMakie.

3 PyPlot

Il package PyPlot è l'interfaccia di Julia per poter utilizzare il sottomodulo `pyplot` del modulo `matplotlib` dell'ecosistema Python.

Matplotlib penso sia, in assoluto, il più completo software disponibile per la grafica dei dati.

E' documentato da un file PDF di quasi 5.000 pagine, non aggiornatissimo, che troviamo all'indirizzo *https://matplotlib.org/2.0.2/Matplotlib.pdf*.

All'indirizzo *https://matplotlib.org/3.5.1/users/index.html* troviamo la stessa documentazione, in linea, aggiornata al momento in cui scrivo (maggio 2022).

Se abbiamo installato il package PyCall possiamo accedere a tutte le funzioni di `matplotlib`.

Installando il package PyPlot possiamo accedere alle funzioni del sottomodulo `pyplot` che qui interessa.

All'indirizzo *https://buildmedia.readthedocs.org/media/pdf/pyplotjl/latest/pyplotjl.pdf* troviamo un manuale di PyPlot che non è gran che ma è sempre meglio della documentazione ermetica che troviamo su GitHub.

Spero che l'uno e l'altra diventino più comprensibili dopo aver letto questo mio tentativo di divulgazione per principianti.

Dal momento che mi rivolgo a principianti limito anche gli argomenti alle tre cose più ricorrenti nella grafica dei dati: corrispondenze sul piano cartesiano, grafici a barre (detti anche a canne d'organo) e grafici a torta.

Per fare queste cose abbiamo tre funzioni, rispettivamente `plot()`, `bar()` e `pie()`.

plot()

La funzione `plot()` funziona esattamente come l'omonima funzione che abbiamo visto parlando del package Winston.

Unica differenza, gli array denominati `x` e `y` da cui partiamo devono essere dei vettori, cioè con elementi tra parentesi quadre, separati da virgola o punto e virgola. La funzione
`plot(x, y)`
ne evidenzia la corrispondenza su un piano cartesiano, con il vettore `x` sulle ascisse e il vettore `y` sulle ordinate.

Per default la corrispondenza viene evidenziata con una linea continua di colore blu.

Possiamo ottenere altri colori inserendo tra le parentesi tonde, dopo i primi due argomenti e separando con una virgola, il parametro opzionale
"<iniziale_colore>"
dove <iniziale_colore> è la lettera iniziale del nome del colore in lingua inglese (`y` yellow, `m` magenta, `c` cyan, `r` red, `g` green, `b` blue, `w` white) con la sola eccezione del nero che viene indicato con `k` per non confonderlo con il blu.

Con lo stesso parametro opzionale possiamo ottenere altri modi di evidenziare le corrispondenze tra i dati utilizzando le seguenti sigle: `-` per la linea continua, `:` per la linea punteggiata, `--` per una linea a trattini, `+` per il relativo simbolo, `o` per un pallocco, `*` per un asterisco, `.` per un punto, `x` per il relativo simbolo a crocetta, `s` per un quadratino, `d` per il quadratino a diamante, ^ per un triangolino, `v` per un triangolino rovesciato, > per un triangolino puntato a destra, < per un triangolino puntato a sinistra.

Nel costruire il grafico abbiamo a disposizione le funzioni

`title("<stringa>")`

`xlabel("<stringa>")`

`ylabel("<stringa>")`

`grid()`

rispettivamente per dare un titolo al grafico, per descrivere l'asse delle ascisse, per descrivere l'asse delle ordinate e per disegnare una griglia.

Possiamo inserire le istruzioni nella REPL e vedremo progressivamente l'effetto che fanno.

Per salvare il grafico che abbiamo costruito abbiamo a disposizione la funzione

`savefig("<stringa_percorso_nome_grafico>")`

con disponibili i formati corrispondenti alle estensioni `.png`, `.jpg`, `.svg` e `.pdf`.

Possiamo altresì radunare le istruzioni in uno script che salviamo con estensione `.jl` per poterlo eseguire quando vogliamo con

`julia` <nome_file>`.jl`

Nello script possiamo inserire l'istruzione

`show()`

per vedere il grafico.

Se vogliamo utilizzare questa istruzione dobbiamo inserirla come ultima dello script, comunque dopo quella per salvare il grafico.

La funzione `plot()` ci serve anche per tracciare i grafici di funzione.

In questo caso il vettore delle x lo costruiamo con la funzione `range()` che ha la seguente sintassi

`range(`<valore_iniziale>`, stop =` <valore_finale>`, length =` <elementi>`)`

dove

<valore_iniziale> indica il punto da cui comincia il grafico,

<valore_finale> indica il punto dove finisce il grafico,

<elementi> indica il numero dei valori equidistanti contenuti tra il valore iniziale e il valore finale.

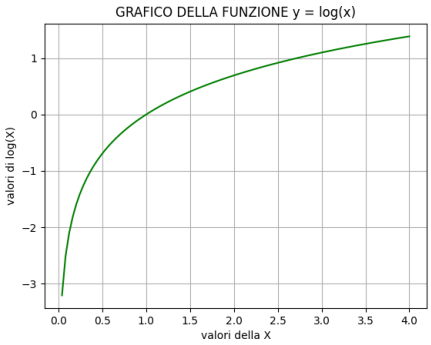
Il vettore per le ordinate possiamo costruirlo scrivendo la funzione con la dot syntax. Cosa molto pericolosa in quanto basta mettere un punto nel posto sbagliato per creare un disastro.

Qui l'unico modo di schivare il pericolo è quello di costruire il vettore delle ordinate ricorrendo alla funzione `map()`.

Qualche esempio per capirci.

```
using PyPlot
title("GRAFICO DELLA FUNZIONE y = log(x)")
xlabel("valori della X")
ylabel("valori di log(X)")
grid()
x = range(0, stop=4, length=100)
y = map(x->log(x), x)
plot(x, y, "g")
savefig("/home/vittorio/Immagini/logaritmo.png")
show()
```

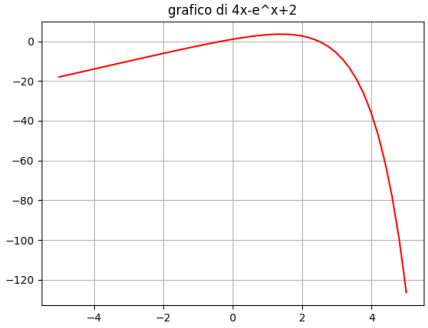
Con lo script ci viene mostrato e viene salvato all'indirizzo indicato il seguente grafico di funzione



Con quest'altro script

```
using PyPlot
title("grafico di 4x-e^x+2")
grid()
x = range(-5, stop=5, length=50)
y = map(x->4*x-exp(1)^x+2, x)
plot(x,y,"r")
savefig("/home/vittorio/Immagini/esponenziale.png")
show()
```

ci viene mostrato e viene salvato all'indirizzo indicato il seguente grafico di funzione



bar()

La funzione `bar()` è deputata alla costruzione di grafici a barre, detti anche a canne d'organo.

I suoi due principali argomenti sono:

. una tupla di stringhe per gli identificatori delle barre,

. un vettore di valori per l'altezza delle barre.

Il primo ad indicare a cosa si riferisce il dato evidenziato da ciascuna barra, il secondo ed evidenziare il valore di quel dato.

I due principali parametri opzionali sono:

. la larghezza della barra, indicata con la sintassi

`width` = <valore>

dove <valore> è compreso tra 0 e 1 (con il valore 1 le barre sono unite tra loro)

. un vettore di stringhe per il colore delle barre indicato con la sintassi

color = <vettore>

dove le stringhe per i colori sono le stesse viste per la funzione `plot()`.

Questo è ciò che basta per creare i nostri grafici a barre da principianti.

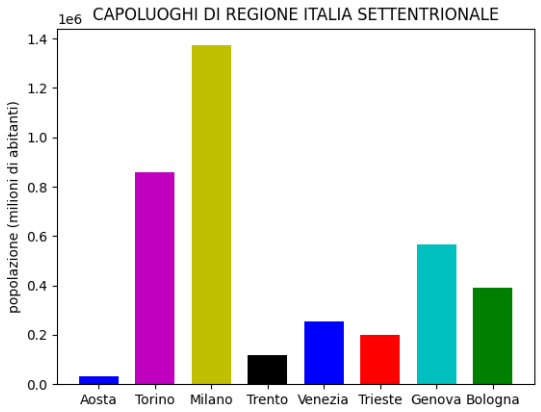
Se vogliamo rendere visibili le differenza tra la popolazione dei capoluoghi delle regioni del Nord Italia che abbiamo nella seguente tabella

Capoluogo	Abitanti
Aosta	33523
Torino	858205
Milano	1374582
Trento	118879
Venezia	256083
Trieste	200609
Genova	566410
Bologna	391686

possiamo farlo con il seguente script

```
using PyPlot
l = ("Aosta","Torino","Milano","Trento","Venezia","Trieste","Genova","Bologna")
x = [33523,858205,1372582,118879,256083,200609,566410,391686]
c = ["b","m","y","k","b","r","c","g"]
ylabel("popolazione (milioni di abitanti)")
title("CAPOLUOGHI DI REGIONE ITALIA SETTENTRIONALE")
bar(l, x, width=0.7, color = c)
savefig("/home/vittorio/Immagini/popolazione.png")
show()
```

che visualizza e salva il seguente grafico



Ovviamente occorre badare a che gli elementi di tuple e array siano ordinati come si deve.

pie()

La funzione `pie()` è deputata alla costruzione di grafici a torta.

L'argomento base da passare alla funzione è:

. un vettore contenente i valori delle fette.

Il principale argomento opzionale è:

. un vettore di stringhe per le etichette delle fette da passare con la sintassi

`labels` = <vettore>

In mancanza di indicazioni al riguardo le fette vengono differenziate con colori scelti a caso.

Se vogliamo scegliere i colori delle fette abbiamo l'argomento opzionale:

. vettore di stringhe per il colore delle fette indicato con la sintassi

`colors` = <vettore>

dove le stringhe per i colori sono le stesse viste per la funzione `plot()`.

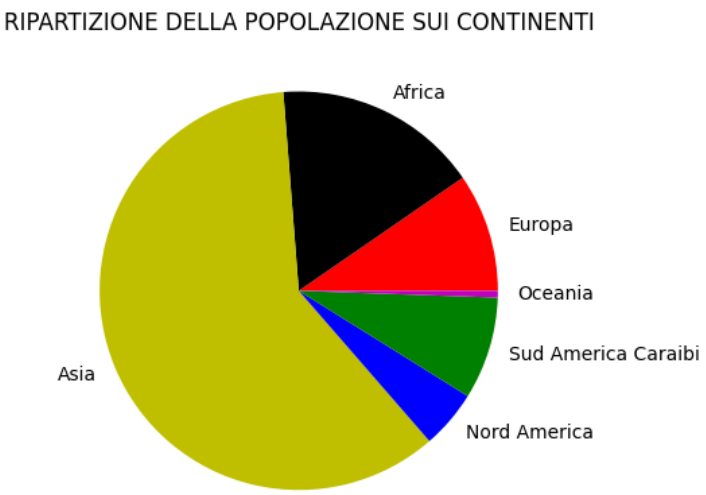
Se vogliamo rendere visibile la ripartizione della popolazione mondiale sui continenti che abbiamo nella seguente tabella

Continente	Popolazione (milioni)
Europa	750
Africa	1.300
Asia	4.700
Nord America	370
Sud America e Caraibi	650
Oceania	43

possiamo farlo con il seguente script

```
using PyPlot
x = [750,1300,4700,370,650,43]
l = ["Europa","Africa","Asia","Nord America", "Sud America Caraibi","Oceania"]
c = ["r","k","y","b","g","m"]
title("RIPARTIZIONE DELLA POPOLAZIONE SUI CONTINENTI")
pie(x, labels = l, colors = c)
savefig("/home/vittorio/Immagini/continenti.png")
show()
```

che visualizza e salva il seguente grafico



Parte II
Grafica per interfaccia utente (GUI)

Sono personalmente convinto che un linguaggio che ha le finalità di Julia può benissimo dare il meglio di sé senza interfacciarsi graficamente con l'utente: più che una semplificazione, per fare certe cose, l'interfaccia grafica può essere un impiccio.

Salvo casi in cui l'interfaccia grafica sia unificata in un'applicazione in cui si accolgano dati dall'utente e si rappresentino in tempo reale gli effetti grafici del cambiamento dei dati. Per cose di questo tipo esiste il pacchetto Makie, cui ho accennato nella Parte precedente, molto impegnativo sia per l'hardware sia per chi lo usa, che lasciamo ad esperti professionisti.

Per dilettanti come noi, fino a qualche tempo fa, Julia prevedeva la possibilità di creare interfacce grafiche utilizzando il pacchetto Tk.jl, che metteva a nostra disposizione tutto l'armamentario del software Tk, come avviene in Python con il modulo tkinter.

Visto che ormai tutti sono convinti che le GUI fatte con Tk siano vecchie e brutte (mentre fanno benissimo il loro mestiere con poca spesa), nel modo Python un volenteroso ha creato customtkinter per far si che le GUI tkinter siano più moderne mentre, nel mondo Julia, è stato deprecato l'uso del pacchetto Tk.

Il fatto è che le alternative, a parte il citato pacchetto Makie, con cui si possono anche realizzare GUI semplici, utilizzando un carro armato per schiacciare una mosca, sono tutte abbastanza complicate e utilizzano le librerie Qt, GTK con pacchetti che troviamo nel repository di Julia (QML.jl, GTK.jl).

A questo punto, per noi dilettanti, la via migliore è quella di ricordarci che da Julia possiamo avere accesso ai moduli Python attraverso il pacchetto PyCall e, per questa via, costruire GUI utilizzando i moduli `tkinter` o `customtkinter` di Python.

Un piccolo esempio.

Con questo script creiamo una finestra utilizzando il tool `customtkinter` di Python.

```
using PyCall
tk = pyimport("customtkinter")
r = tk.CTk()
r.title("PROVA")
r.config(width = 300, height = 200, bg = "yellow")
r.mainloop()
```

con questo risultato

Attenzione al fatto che lo script deve girare su un interprete/compilatore Julia, pertanto i doppi apici, che se fossimo in Python potrebbero anche essere semplici apici, qui devono essere proprio doppi (in Julia, infatti, tra semplici apici ci può stare un solo carattere, non una stringa).

Con quest'altro script creiamo una piccola applicazione che chiede all'utente il nome per salutarlo

```
using PyCall
function saluta()
nome = e.get()
l2.configure(text="Ciao, " * nome * "!")
end
```

```
tk = pyimport("customtkinter")
r = tk.CTk()
r.title("SALUTO")
r.config(bg = "PaleGreen")
l1 = tk.CTkLabel(r, text = "Come ti chiami?", fg_color = "PaleGreen")
l1.grid(column = 0, row = 0, sticky = "w", padx = 4)
e = tk.CTkEntry(r)
e.grid(column = 1, row = 0, padx = 4, pady = 4)
b = tk.CTkButton(r, text = "SALUTA", width = 80, command = saluta)
b.grid(column = 0, row = 1, columnspan = 2, pady = 4)
l2 = tk.CTkLabel(r, height = 30, text = "", fg_color = "PaleGreen", text_color = "red", font = ("Amigos",20))
l2.grid(column = 0, row = 2, columnspan = 2, pady = 4)
r.mainloop()
```

Come si vede, anche scrivendo istruzioni tipiche del modulo customtkinter si deve rispettare la sintassi di Julia (oltre all'attento uso del doppio apice per le stringhe abbiamo l'operatore * per concatenare stringhe).

Il risultato è il seguente

A sinistra l'applicazione all'apertura e a destra l'applicazione dopo aver inserito il nome della persona da salutare e cliccato sul pulsante SALUTA.