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, 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 (maggio 2022) 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

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 Winston

Winston è un package originale dell'ecosistema Julia.

Si presta soprattutto per grafici lineari ed è pertanto particolarmente adatto per i grafici di funzione in matematica.

La documentazione che troviamo all'indirizzo *https://winston.readthedocs.io/en/latest/* non è gran che. Nel momento in cui scrivo (maggio 2022) è introdotta dal messaggio «Please pardon our dust! Docs under construction».

Per fare la cosa che viene meglio con Winston, ci bastano comunque poche cose e le possiamo anche sperimentare lavorando nella REPL di Julia inserendo via via i comandi che vedremo.

Partiamo con il comando

```
using Winston
```

In presenza di due array monodimensionali (elementi tra parentesi quadre, separati da spazio, virgola o punto e virgola), denominati `x` e `y`, abbiamo la funzione

```
plot(x, y)
```

che 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 nero.

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 punto e trattino alternati, `--` 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.

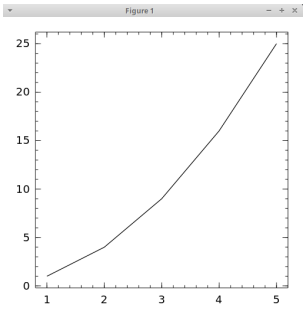
In presenza di questi due array

```
x = [1 2 3 4 5]
y = [1 4 9 16 25]
```

scrivendo nel REPL di Julia, dopo aver scritto `using Winston`,

`plot(x, y)`

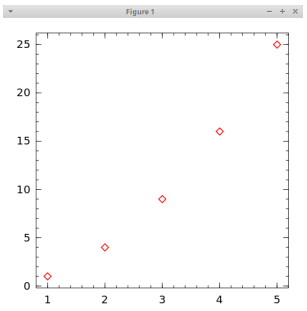
otteniamo il grafico



e scrivendo

`plot(x, y, "rd")`

otteniamo il grafico



Se vogliamo conservare il grafico, subito dopo che l'abbiamo ottenuto scriviamo

`savefig("<percorso_e_nome_file>")`

dove al nome file possiamo dare estensione `.png`, `.eps`, `.pdf` o `.svg` ottenendo file nei relativi formati grafici.

Con

`oplot(x1, y1)`

possiamo sovrapporre un altro grafico a quello precedentemente ottenuto con `plot()` e possiamo farlo quante volte vogliamo: gli assi e i grafici si adatteranno in modo da evidenziare correttamente le corrispondenze.

Nel costruire il grafico abbiamo a disposizione le funzioni

`title("<stringa>")`
`xlabel("<stringa>")`
`ylabel("<stringa>")`

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

Ovviamente se scriviamo le istruzioni una via l'altra in un file di testo e lo salviamo con estensione `.jl` otteniamo uno script rieseguibile scrivendo a terminale

`julia` <nome_file>`.jl`

In questo modo però non possiamo avere le anteprime dei nostri grafici, come avviene se lavoriamo sulla REPL, e dobbiamo concludere il nostro script con l'istruzione di salvataggio dei grafici stessi per poterli vedere.

* * *

La funzione `plot()` può essere utilizzata anche per i grafici di funzioni matematiche.

Per ottenere questo dobbiamo avere un array per l'asse delle ascisse (x) con valori abbastanza ravvicinati in modo che la linea descrittiva della funzione sia liscia e continua e dobbiamo avere un array per l'asse delle ordinate (y) che rappresenta tutti i corrispondenti valori della funzione.

L'array per le ascisse possiamo costruirlo con la funzione `range()` che ha la seguente sintassi
`range(<valore_iniziale>, stop = <valore_finale>, length = <elementi>)`
dove
<valore_iniziale> indica il valore dell'ascissa da cui comincia il grafico,
<valore_finale> indica il valore dell'ascissa dove finisce il grafico,
<elementi> indica il numero dei valori equidistanti contenuti tra il valore iniziale e il valore finale.

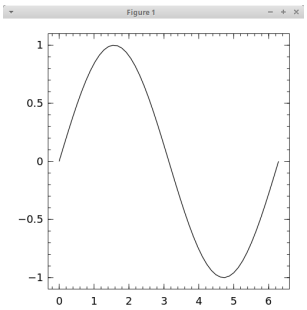
L'array per le ordinate possiamo costruirlo scrivendo la funzione con la dot syntax.

Un esempio per capirci.

Proponiamoci di creare il grafico della funzione $y = sin(x)$ per l'intervallo tra 0 e 2π.
Nella REPL di Julia scriviamo, uno dopo l'altro, i seguenti comandi

```
using Winston
x = range(0, stop=2*pi, length=50)
y = sin.(x)
plot(x, y)
```

e potremo apprezzare l'anteprima



Il parametro `length` passato alla funzione `range()` deve essere abbastanza elevato per rendere liscia e continua la linea del grafico: come si vede anche con il modesto valore di `50` il risultato è buono.

Quando la funzione è rappresentata da un'espressione matematica complessa risulta scomodo renderla con la dot syntax.
Per esempio, la funzione $y = 4x^3 - 2x^2 + 1$ dovrebbe essere scritta così
`y = x .^ 3 * 4 - x .^ 2 * 2 .+ 1`

A semplificare le cose interviene la possibilità di produrre i grafici delle funzioni matematiche utilizzando la funzione `fplot()` alla quale passiamo come parametri la funzione, in forma anonima, di cui produrre il grafico e l'intervallo di interesse, con la sintassi
`fplot(<funzione_anonima>, [<inizio>,<fine>])`
con, in più, l'eventuale parametro opzionale per il colore nel modo visto prima.

Così, con la semplice istruzione

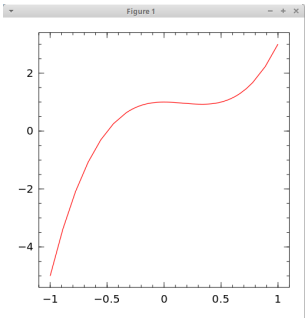
```
fplot(x->sin(x), [0, 2*pi], "y")
```

otteniamo lo stesso grafico di prima tracciato in colore giallo.

Con

```
fplot(x->4*x^3-2*x^2+1, [-1,1], "r")
```

otteniamo il grafico



Nel manualetto `julia`, citato in premessa, ho proposto il seguente script per il calcolo della regressione tra due serie di dati

```
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)
```

Lanciandolo, dopo avere inserito i dati tra cui verificare se esiste correlazione, ci vengono visualizzati a terminale
. i dati che abbiamo inserito,
. il coefficiente della correlazione esistente tra loro,
. l'espressione della retta di regressione,
. il coefficiente di determinazione.

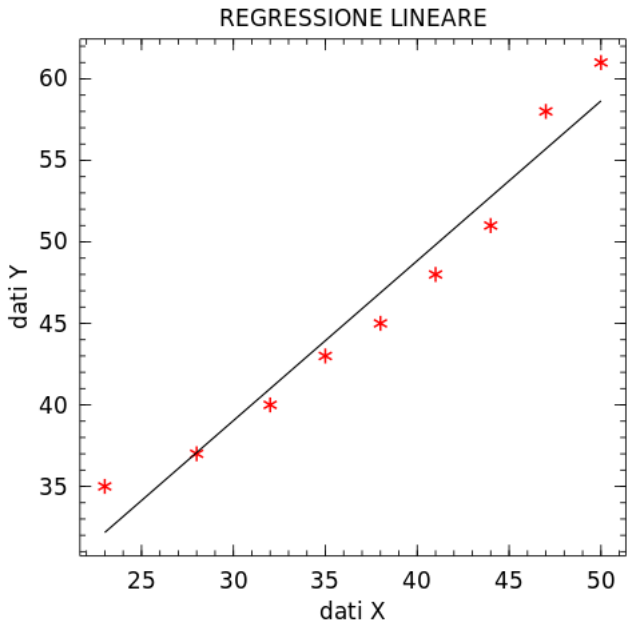
Una cosa così

```
dati:
23.0 35.0
28.0 37.0
32.0 40.0
35.0 43.0
38.0 45.0
41.0 48.0
44.0 51.0
47.0 58.0
50.0 61.0

Coefficiente di correlazione:
0.9759247340771222

Retta di regressione:
y = 9.621169916434532 + 0.9805013927576604x

Coefficiente di determinazione:
0.9524290865835017
```

Se aggiungiamo allo script quanto segue

```
using Winston
plot(lx, ly, "r*")
x1 = range(minimum(lx), stop = maximum(lx), length = 50)
y1 = b .* x1 .+ a
oplot(x1, y1)
xlabel("dati X")
ylabel("dati Y")
title("REGRESSIONE LINEARE")
savefig("/home/vittorio/Immagini/regressione.png")
```

ritroveremo all'indirizzo indicato in quest'ultima riga il file `regressione.png` così concepito



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.

Come praticamente tutti i packages grafici originali dell'ecosistema Julia fa tantissime cose ma non ci consente di fare cose semplici come un grafico a torta di quelli che possiamo fare con Excel o Calc di LibreOffice.

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 stessa funzione `range()` che abbiamo utilizzato nel caso del package Winston e 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.

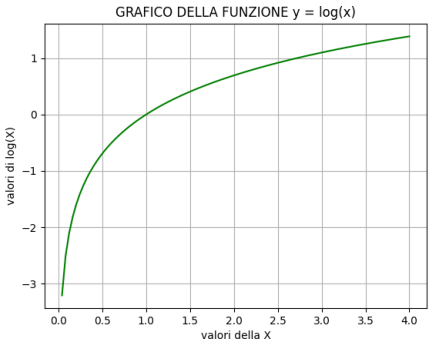
Il vettore per le ordinate possiamo costruirlo scrivendo la funzione con la dot syntax, come abbiamo visto fare quando abbiamo parlato del package Winston. Cosa molto pericolosa in quanto basta mettere un punto nel posto sbagliato per creare un disastro.

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()
```

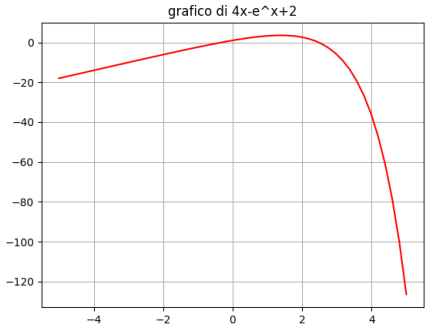
Con lo script ci viene mostrato e viene salvato all'indirizzo indicato il seguente grafico di funzione



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.

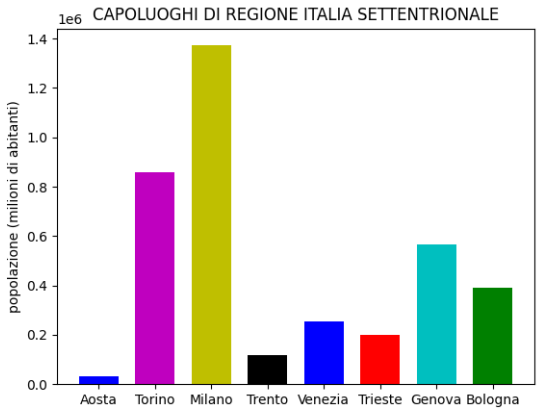
Se vogliamo rendere visibili le differenza tra la popolazione dei capoluoghi delle regioni del Nord Italia che abbiamo nella seguente tabella

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()`.

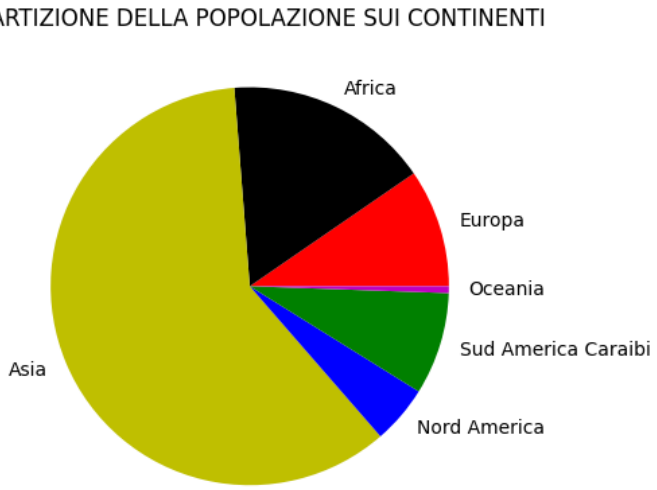
Se vogliamo rendere visibile la ripartizione della popolazione mondiale sui continenti che abbiamo nella seguente tabella

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 non abbia nessun bisogno di interfacciarsi graficamente con l'utente: più che una semplificazione, per fare certe cose, considero l'interfaccia grafica un impiccio.

Tuttavia, tra i package che arricchiscono l'ecosistema Julia ne abbiamo che consentono di costruire GUI, in particolare utilizzando il framework GTK oppure il più anziano e glorioso framework Tk.

Per quest'ultimo nutro particolare simpatia[1] e qui parlerò del package Julia che lo interfaccia e che si chiama appunto Tk.

4 Tk

E' un package originale dell'ecosistema Julia che, in modo semplificato, sia sul piano della facilità di uso sia sul piano dell'estetica dei risultati, con pochissima fatica ci consente di arricchire, per chi lo considera un arricchimento, i nostri script Julia.

Le semplificazioni sul piano estetico non ci consentono di usare i colori, di formattare le scritte e di utilizzare altre opzioni di abbellimento che ritroviamo nel Tk originale aggregato al linguaggio Tcl e nel suo interfacciamento per il linguaggio Python, denominato Tkinter.

All'indirizzo *https://github.com/JuliaGraphics/Tk.jl/blob/master/docs/src/index.md* si trova il massimo disponibile di documentazione su questo pacchetto e, nello stile GitHub, si tratta di una cosa ermetica per addetti ai lavori.

Cerco qui di radunare le cose essenziali ad uso di principianti dilettanti.

La prima cosa di cui abbiamo bisogno è la finestra radice, quella su cui si costruisce tutto il resto. Lavorando con Tk di Julia, dopo aver dichiarato

```
using Tk
```

dandogli il nome `r`, la costruiamo con

```
r = Toplevel()
```

potendo mettere tra le parentesi tonde, nell'ordine, tre argomenti facoltativi:
. una stringa contenente il titolo da dare alla finestra,
. un numero indicante la larghezza della finestra in pixel,
. un numero indicante l'altezza della finestra in pixel.
Per il nome della finestra possiamo sbizzarrirci come vogliamo, a me piace `r`, che sta per root (radice).
Il titolo dovremmo sceglierlo in sintonia con il contenuto dello script che costruiremo.
Con larghezza e altezza forziamo le dimensioni minime che assumerà il frame quando lo creeremo. Se non le indichiamo, il frame che inseriremo avrà dimensioni totalmente elastiche.

Se lavoriamo nella REPL e vogliamo evitare che la finestra scompaia visivamente al primo inserimento di un frame vuoto, facciamo seguire l'istruzione

```
pack_stop_propagate(r)
```

Widget contenitori

Come avviene per Tcl/Tk e per Tkinter, anche in Tk Julia sono contemplati due widget contenitori, il canvas e il frame. L'unico che può contenere altri widget che siano i veri protagonisti della GUI è il frame. Il canvas diventa qui utile solo per ospitare elementi generati da altri package e Tk Julia non contempla istruzioni per lavorarci, pertanto in questa sede non ci interessa.

Frame

Il frame, se lo chiamiamo `f`, si costruisce con

```
f = Frame(<nome_finestra_radice>)
```

e si colloca nella finestra radice con

```
pack(<nome_frame>, expand = true, fill = "both")
```

dove, con le opzioni `expand` e `fill` messe in questo modo scegliamo che il frame si estenda su tutta la finestra radice e la copra di un colore base sia orizzontalmente che verticalmente.

[1] Ho parlato del framework Tk sul mio blog all'indirizzo *www.vittal.it* nell'allegato in formato PDF «tkinter» all'articolo «Grafica con Python» del maggio 2018 e nell'allegato in formato PDF «tcl_tk» all'articolo «Tcl/tk: un tesoro nascosto» del dicembre 2019. Gli stessi contenuti si trovano, rispettivamente, nei miei manualetti «Tutto Python per principianti» e «Programmare con Tcl/Tk» che si trovano in formato cartaceo su Amazon.

* * *

Se diamo queste istruzioni nella REPL, una dopo l'altra

```
using Tk
r = Toplevel("PROVA", 300, 150)
pack_stop_propagate(r)
f = Frame(r)
pack(f, expand = true, fill = "both")
```

ci troviamo di fronte questo

Widget indispensabili per costruire una GUI

Qualsiasi interfaccia utente deve necessariamente ed almeno consentire all'utente di comunicare con il computer, al computer di comunicare con l'utente ed all'utente di far partire o di arrestare una o più attività del computer.

Per poter fare queste cose Tk ci mette a disposizione tre widget: uno per fornire dati al computer (che viene chiamato entry), uno per leggere dati che ci restituisce il computer o per rendere visibili istruzioni per l'utente (che viene chiamato label) ed uno per dare il via o per stoppare determinate azioni del computer (che viene chiamato button).

Entry

Il widget entry consiste in una finestrella che serve per immettere una singola linea di testo.

Chiamandolo `e` (o come si vuole), si costruisce con il comando

```
e = Entry(<nome_frame>)
```

Tra le parentesi tonde, dopo il nome del frame e separando con una virgola, possiamo indicare l'opzione

```
width = <n>
```

dove <n> è l'ampiezza della finestrella in numero di caratteri.

Label

Il widget label consiste in uno spazio per esporre testo e numeri. E' utile sia per scrivere nella più ampia finestra della GUI istruzioni e avvertenze per l'utente sia per scrivere le risposte del calcolatore alle elaborazioni che abbiamo chiesto.

Chiamandola `l` (o come si vuole) si costruisce con il comando

```
l = Label(<nome_frame>)
```

Tra le parentesi tonde, dopo il nome del frame e separando con una virgola, possiamo inserire una stringa (tra doppi apici) contenente il testo eventualmente da inserire alla sua creazione.

Dal momento che il frame contenitore di Tk è totalmente elastico, ad evitare che il contenuto di una Label troppo lungo allarghi il frame stesso fino a debordare dallo schermo, nel costruire la Label possiamo inserire l'opzione

```
wraplength = <larghezza_massima>
```

dove <larghezza_massima> è il numero di pixel di larghezza massima della finestra, raggiunto il quale il contenuto della Label va a capo.

Button

Il widget button è un pulsante sul quale si clicca con il mouse per far fare qualche cosa al computer.

Chiamandolo `b` (o come si vuole) si costruisce con il comando
`b = Button(`<nome_frame>, <stringa_etichetta>`)`
dove <stringa_etichetta> è, tra doppi apici, ciò che va scritto sul pulsante ad indicarne la finalità.

Se nello script è già stata creata la funzione da richiamare con il click possiamo semplicemente inserirne il nome come terzo argomento del costruttore perché essa venga richiamata.

* * *

Esistono due importanti funzioni che hanno a che fare con il contenuto dei widget che abbiamo visto:
`get_value(`<nome_widget>`)` acquisisce il contenuto del widget come stringa,
`set_value(`<nome_widget>`,` <stringa>`)` inserisce una stringa nel widget.

Disposizione dei widget

Per la collocazione dei widget nel frame, Tk Julia ci mette a disposizione due delle tre geometrie tradizionali di Tk: pack e grid.

Geometria con la modalità pack

La funzione per inserire i widget è `pack()`, il cui primo argomento è il nome del widget seguito dalle eventuali opzioni.

Nell'impostazione di default i widget vengono inseriti impacchettati uno via l'altro, dall'alto in basso: il primo inserito sta in alto, il secondo sta sotto di lui, ecc.

Possiamo modificare questa impostazione con l'opzione `side =`, che accetta i valori precostituiti `left` , `right` , `bottom` e `top` (quello di default), da indicare tra doppi apici.

Per comprendere l'effetto di queste opzioni occorre considerare che, nella geometria pack, il contenitore, salvo indicazione di una dimensione minima, è elastico e, quando vi si inserisce il primo widget, esso si restringe attorno ad esso. Ciò non toglie che la cavità in cui inserire gli altri widget, anche se non si vede più, rimanga a disposizione.

Nell'impostazione di default la cavità disponibile si sviluppa sotto il widget inserito, in quanto questo è nella posizione top, per cui il successivo inserimento si collocherà sotto il primo widget. Se avessimo inserito il primo widget con l'opzione side a valore bottom, la cavità libera si svilupperebbe verso l'alto, sopra il primo widget inserito, e il successivo si collocherebbe ancora sopra.

In entrambi i casi la cavità occupata si sviluppa sia a sinistra sia a destra del widget inserito e sopra (nel caso di top) o sotto (nel caso di bottom), lasciando libera tutta la fascia orizzontale rispettivamente sottostante o sovrastante in cui poter inserire altri widget scegliendo tra sinistra e destra.

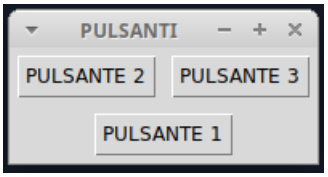
Purtroppo quando scegliamo uno dei valori left o right per l'opzione side, la cavità libera rimane, rispettivamente, a destra o a sinistra del widget e, se dopo aver inserito, per esempio, un widget con il valore left assegnato all'opzione side, ne inseriamo uno con il valore top assegnato all'opzione side ce lo troveremo sì al top, ma nella parte destra della finestra e mai più, con la geometria pack nello stesso contenitore, potremo inserirlo al top e al centro. Con questa geometria siamo pertanto costretti a ricorrere spesso a sdoppiamenti dei contenitori, sapendo tuttavia che il contenitore radice è uno e non sdoppiabile.

Altre opzioni ci consentono di distanziare tra di loro i widget in modo che non si presentino troppo attaccati tra di loro. Si tratta delle opzioni `padx =`, cui possiamo assegnare il valore in pixel per i distanziamenti orizzontali e `pady =`, cui possiamo assegnare il valore in pixel per i distanziamenti verticali.

Con questo script

```
using Tk
r = Toplevel("PULSANTI")
f = Frame(r)
pack(f, expand=true, fill="both")
p1 = Button(f, "PULSANTE 1")
pack(p1, side="bottom", padx=5, pady=5)
p2 = Button(f, "PULSANTE 2")
pack(p2, side="left", padx=5, pady=5)
p3 = Button(f, "PULSANTE 3")
pack(p3, side="right", padx=5, pady=5)
readline()
```

otteniamo una disposizione di questo tipo



PULSANTE 1 è stato inserito con l'opzione `bottom` in modo che i successivi vengano inseriti sopra di esso. Per PULSANTE 2 si è scelto di inserirlo sulla sinistra e per PULSANTE 3 si è scelto di inserirlo sulla destra. Senza queste scelte essi sarebbero stati inseriti uno sopra l'altro, sempre al di sopra di PULSANTE 1.

L'istruzione `readline()` finale serve a mantenere visibile la finestra in attesa che si prema un tasto per chiuderla; in mancanza di ciò la finestra scomparirebbe subito.

Geometria con la modalità grid

Anche nella geometria grid il contenitore è elastico e si restringe attorno ai widget man mano li inseriamo.

Il grande vantaggio del metodo sta nel fatto che la cavità nella quale inseriamo i widget è idealmente divisibile in righe e colonne, a mo' di griglia (grid, appunto), numerate da 1 in su con l'origine in alto a sinistra.

La funzione per inserire i widget è `grid()`, il cui primo argomento è il nome del widget, seguito dal numero della riga e dal numero della colonna in cui collocarlo.

Se l'inserimento deve avvenire a cavallo di più righe, al posto del numero di riga indichiamo i numeri delle righe che delimitano lo spazio su cui sovrapporre divisi dal segno di due punti (`:`).

Se l'inserimento deve avvenire a cavallo di più colonne, facciamo la stessa cosa con i numeri delle colonne che delimitano lo spazio su cui sovrapporre.

Ancora di seguito possiamo indicare le opzioni `padx=` e `pady=` per di distanziamenti.

In modalità grid possiamo ottenere la stessa disposizione di pulsanti di prima con il seguente script:

```
using Tk
r = Toplevel("PULSANTI")
f = Frame(r)
pack(f, expand=true, fill="both")
p1 = Button(f, "PULSANTE 1")
grid(p1, 2, 1:2, padx=5, pady=5)
p2 = Button(f, "PULSANTE 2")
grid(p2, 1, 1, padx=5, pady=5)
p3 = Button(f, "PULSANTE 3")
grid(p3, 1, 2, padx=5, pady=5)
readline()
```

Gestione degli eventi

Una delle caratteristiche fondamentali di un programma con interfaccia grafica è quella di poter indurre il computer a fare una certa cosa in corrispondenza ad una certa azione compiuta dall'utente sull'interfaccia grafica, azione che in informatica si usa chiamare evento: si può trattare del click su un pulsante, della pressione di un particolare tasto sulla tastiera mentre si è posizionati su un certo widget o del passaggio del mouse in una certa area dello schermo, ecc.

Gli eventi gestiti da Tk Julia sono due: il click su un widget Button e la pressione del tasto INVIO della tastiera in corrispondenza di un widget Entry.

Per quanto riguarda il click sul Button abbiamo già visto che, se la funzione da attivare è già stata creata basta che ne inseriamo il nome nel costruttore del Button e quanto previsto dalla funzione sarà eseguito in conseguenza del click.

In ogni caso esiste la funzione `bind()` con le seguente sintassi:
`bind(<nome_button>, "command", <nome_funzione>)`
`bind(<nome_entry>, "<Return>", <nome_funzione>)`

per la gestione, rispettivamente, degli eventi click su pulsante e pressione del tasto INVIO nella finestrella di immissione del testo.

Ciò che dovrà essere eseguito in corrispondenza dell'evento viene programmato in una funzione secondo la sintassi del linguaggio Julia
`function` <nome_funzione> (ok)
<istruzioni>
`end`
Trattandosi di una funzione di callback, deve avere almeno un argomento (in mancanza di meglio suggerisco la parola ok).

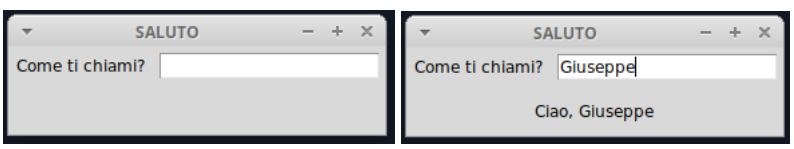
Qualche esempio

A questo punto abbiamo tutti gli elementi per creare script dotati di GUI per fare le cose più ricorrenti.

Un banalissimo esempio di domanda/risposta lo troviamo in questo script

```
using Tk
w = Toplevel("SALUTO")
f = Frame(w)
pack(f, expand=true, fill="both")
l1 = Label(f, "Come ti chiami?")
grid(l1, 1, 1, padx=5, pady=5)
e = Entry(f)
grid(e, 1, 2, padx=5, pady=5)
l2 = Label(f)
grid(l2, 3, 1:2, padx=5, pady=10)
function saluta(ok)
    nome = get_value(e)
    messaggio = "Ciao, $nome"
    set_value(l2, "$messaggio")
end
bind(e, "<Return>", saluta)
readline()
```

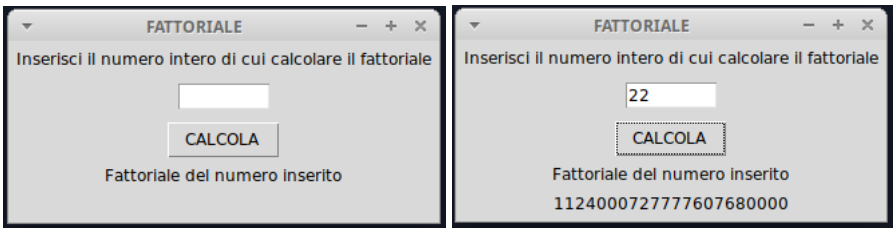
Sulla sinistra abbiamo la GUI generata dallo script e sulla destra abbiamo la stessa GUI dopo che è stato inserito il nome della persona da salutare e premuto il tasto INVIO.



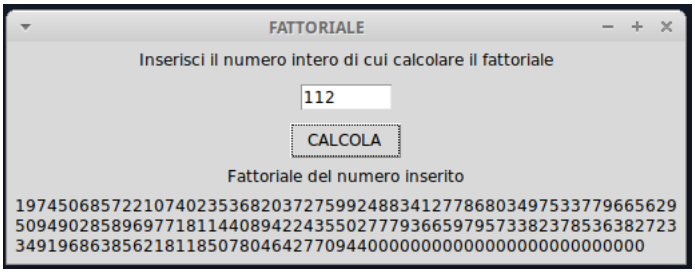
Con quest'altro script calcoliamo il fattoriale di un numero

```
using Tk
function calcola(ok)
    numero = get_value(e)
    n = parse(Int, numero)
    fattoriale = factorial(BigInt(n))
    set_value(l3, string(fattoriale))
end
w = Toplevel("FATTORIALE")
f = Frame(w)
pack(f, expand=true, fill="both")
l1 = Label(f, "Inserisci il numero intero di cui calcolare il fattoriale")
pack(l1, padx=5, pady=5)
e = Entry(f, width = 8)
pack(e, padx=5, pady=5)
b = Button(f, "CALCOLA", calcola)
pack(b, padx=5, pady=5)
l2 = Label(f, "Fattoriale del numero inserito")
pack(l2)
l3 = Label(f,wraplength=500)
pack(l3, padx=5, pady=5)
readline()
```

Sulla sinistra abbiamo la GUI generata dallo script e sulla destra abbiamo la stessa GUI dopo che è stato inserito un numero di cui calcolare il fattoriale e premuto il pulsante CALCOLA con un risultato normalmente inseribile nella GUI stessa



Qui abbiamo la GUI come si è allargata per contenere un risultato di dimensioni superiori alla GUI generata dallo script, rispettando comunque la dimensione massima di 500 pixel imposta alla Label che ospita il risultato stesso attraverso l'opzione `wraplenth`



* * *

A questo punto penso che il lettore abbia le basi per creare GUI anche più complesse di quelle esemplificate e, soprattutto, possa con maggior cognizione di causa appropriarsi del contenuto del documento illustrativo che si trova all'indirizzo
https://github.com/JuliaGraphics/Tk.jl/blob/master/docs/src/index.md
dove è possibile conoscere molti altri widget che ci offre il package Tk.