UNIVERSITÀ DEGLI STUDI ROMA TRE

SCUOLA DI ECONOMIA E STUDI AZIENDALI

DIPARTIMENTO DI ECONOMIA

CORSO DI LAUREA TRIENNALE IN ECONOMIA

"*MODELLI MATEMATICI PER L'ECONOMIA CON I LINGUAGGI R, OCTAVE E PYTHON*"

LAUREANDO

FEDERICO DE ANGELIS SCORSONE

MATRICOLA N. 520317

RELATRICE

PROF.SSA VALENTINA GUIZZI

ANNO ACCADEMICO 2019-2020

INDICE

Introduzione

La '*matematizzazione*' della teoria economica comincia a partire dagli anni '70 del XIX secolo.

In questa precisa fase il pensiero economico muta improvvisamente. L'obiettivo principale degli economisti non è più analizzare quali siano le condizioni che permettano il corretto e continuo funzionamento[1] di un'economia (riflessione che li ha tenuti impegnati per i 100 anni precedenti[2]), bensì identificare e definire quale sia l'utilizzo *ottimale* delle risorse quando esse siano scarse, al fine di soddisfare i bisogni ed i desideri dei soggetti economici.

Tale visione continua ad essere molto presente. Spesso abbiamo infatti abbiamo sentito spesso parlare di funzioni di utilità, funzioni di produzione, funzioni di consumo. Queste non sono altro che rappresentazioni matematiche di processi economici, e talvolta sociali o finanziari, che permettono agli utilizzatori di modellizzare (o provare a farlo) problemi reali, ipotizzati o teorici.

Questa sostanziale trasformazione della teoria economica subisce limature di forma con l'avvento dei calcolatori. Grazie alla loro potenza di calcolo l'impiego è immediato in processi simulativi o concernenti l'elaborazione di dati empirici, riconducibili in ultima istanza a finalità teoriche.

In questo lavoro ci occuperemo esattamente di questo passaggio: la nostra indagine consisterà nell'individuare alcuni iniziali problemi matematici, strettamente collegati all'economia, per formalizzarli attraverso diversi linguaggi di programmazione.

---

[1] Laddove con 'funzionamento' si intende il processo comprendente la produzione, la distribuzione, l'accumulazione e la circolazione del prodotto

[2] Ovviamente con ciascuno le diverse sfumature

Se nel XIX secolo la matematica è diventata essenziale all'interno della teoria economica, oggi è altresì importante avere dimestichezza degli strumenti informatici, ormai insostituibili, per comprendere ed analizzare ulteriormente i processi economici, sociali e finanziari.

La finalità di questa tesina è pertanto quella di tradurre in linguaggio di programmazione alcuni semplici modelli matematici in economia. Nel dettaglio ci occuperemo quindi di formalizzare nei diversi linguaggi le rappresentazioni grafiche delle funzioni, la ricerca di massimi e minimi e problemi di ottimizzazione.

I linguaggi di programmazione

In questa tesi esamineremo tre linguaggi di programmazione: ***R***, ***Python*** e ***Octave***.

R è un linguaggio di programmazione per il calcolo statistico e la rappresentazione grafica, uscito per la prima volta nel 1993. Sebbene sia pensato principalmente per l'utilizzo statistico, esso presenta pacchetti aggiuntivi e funzioni integrate utili anche per problemi matematici. Useremo l'interfaccia grafica RStudio.

Octave è un linguaggio di programmazione principalmente pensato per il calcolo numerico e scientifico. Le sue applicazioni sono molteplici ed è molto simile a MATLAB per la sintassi ed i comandi utilizzati. Semplificando, si tratta del suo corrispondente *open-source*, in quanto condivide gran parte della sintassi.

Python invece è un linguaggio di programmazione del 1991. Negli ultimi anni grazie alla sua semplicità e flessibilità, oltre che per le innumerevoli librerie aggiuntive, è stato molto utilizzato per il calcolo scientifico e la scienza dei dati. Utilizzeremo l'IDLE[3] fornito di default con l'installazione.

[3] Integrated Development and Learning Environment

All'interno del nostro lavoro non ci soffermeremo molto sui dettagli tecnici di ciascun linguaggio, analizzabili dalla documentazione presente in rete, ma formalizzeremo direttamente i problemi matematici, arricchendoli con spunti e notazioni.

Funzioni di una variabile

Definizione e grafici

Introduciamo alcune nozioni di base sulle funzioni. Si dice funzione una legge che assegna ad ogni elemento x di un insieme A uno e un solo elemento y di un insieme B e si scrive $f: A \to B$. Una funzione mette in relazione pertanto gli elementi di due insiemi.

Indicato con $f(x)$ l'elemento dell'insieme B che è assegnato ad un elemento x dell'insieme A attraverso la funzione f, l'insieme A degli elementi su cui f è definita è detto *dominio* di f.

Le funzioni reali di variabile reale le possiamo indicare mediante la seguente notazione:

$$f: A \subseteq \mathbb{R} \to \mathbb{R} \text{ dove } \mathbb{R} \text{ è l'insieme dei numeri reali}$$

Per descrivere accuratamente la realtà è tuttavia inevitabile utilizzare funzioni di più di una variabile. Basti pensare ad esempio ad una funzione di produzione o di utilità.

È necessario quindi generalizzare la precedente nozione di funzione.

Ipotizzando una funzione di produzione con n fattori produttivi $x_1, x_2, \ldots, x_n$ e y la quantità prodotta del bene, si introduce una funzione del tipo $y = f(x_1, x_2, \ldots, x_n)$, ovvero $f: A \subseteq \mathbb{R}^n \to \mathbb{R}$. È ben noto che il grafico di una tale funzione sia un sottoinsieme di insieme $\mathbb{R}^{n+1}$.

Generalizzando ulteriormente, possiamo introdurre funzioni di produzione $f: A \subseteq \mathbb{R}^n \to \mathbb{R}^m$, tali da descrivere l'attività di un'impresa che produce m beni con n fattori produttivi.

Considerando, ad esempio, il caso di un'azienda che produce 2 beni con 3 fattori produttivi. Possiamo descrivere il processo con una funzione del tipo

$$Q = (q_1, q_2) = (f(x_1, x_2, x_3), f_2(x_1, x_2, x_3))$$

con $q_1 = f_1(x_1, x_2, x_3)$ e $q_2 = f_2(x_1, x_2, x_3)$.

Nella teoria d'impresa le funzioni di produzione sono il fondamento analitico. Nel caso della produzione di 1 bene q con 2 fattori produttivi individuiamo:

- funzioni lineari: $q = a_1 x_1 + a_2 x_2$;
- funzioni Cobb-Douglas: $q = k\, x_1^{a} x_2^{b}$;
- funzioni di Leontief: $q = min\left[\frac{x_1}{c_1}, \frac{x_2}{c_2}\right]$;
- funzioni a elasticità costante: $q = k\,(c_1 x_1^{-a} + c_2 x_2^{-a})^{-b/a}$.

Ma come si traducono in informatica le funzioni matematiche?

Prendiamo una funzione, ad esempio $\frac{\sqrt{1+x^2}}{ln(x^3)}$(ossia $\frac{\sqrt{1+x^2}}{3\,ln(x)}$), e formalizziamola nei diversi linguaggi di programmazione. Per verificare la correttezza della scrittura chiederemo ogni volta di calcolare $f(2)$; in questo modo saremo sicuri sulla sintassi utilizzata.

Su ***Octave***, una funzione da $\mathbb{R}$ a $\mathbb{R}$ si formalizza nel seguente modo:

```
funzione1= @(x) (sqrt(1+x.^2))/(log(x.^3))
funzione1(2)
% il comando @(x) associato all'uguaglianza definisce una
%funzione
% il comando sqrt(x) indica una radice quadrata con argomento
x
% l'esponente su octave si identifica con il punto seguito
dall'accento circonflesso
% il logaritmo naturale 'ln' lo indichiamo con la funzione
predefinita log(x)
```

Per quanto riguarda ***R***, possiamo formalizzare la medesima funzione attraverso il comando:

```
funzione1 <- function(x) (sqrt(1+x^2))/log(x^3)
funzione1(2)
#la funzione la definiamo con il comando function;
#dalle documentazioni è preferibile assegnare il nome della
#variabile con una freccia
#le restanti notazioni sono uguali a octave eccetto per gli
#esponenziali.
```

e con ***Python***, dove la situazione è leggermente più complicata:

```
import math
def funzione1(x):
    y=math.sqrt(1+x**2)/math.log(x**3)
```

```
    return y
print (funzione1(2))
#è necessario importare il modulo preinstallato con
#all'interno le funzioni matematiche. definire una funzione
#(che in questo caso ha una variabile x ed è una funzione
#matematica come la intendiamo noi) e chiuderla con return.
#Poi digitiamo il comando print[4] per chiedergli quanto vale
#f(2).
```

Per funzioni da $\mathbb{R}^2$ a $\mathbb{R}$, la questione varia di poco. Consideriamo una funzione $z = f(x, y)$ definita come $Z = \frac{\sqrt{x^3}}{y+2}$; su ***Octave*** diventa:

```
funzione2=@(x,y) (sqrt(x.^3))/(y+2)
funzione2(2,2)
```

su ***R*** invece:

```
funzione2 <- function(x,y) (sqrt(x^3))/(y+2)
funzione2(2,2)
```

Su ***Python*** basta aggiungere:

```
def funzione2(x,y):
    z=math.sqrt(x**3)/y+2
    return z
print (funzione2(2,2))
```

Una funzione, oltre ad essere calcolata in un punto, può anche essere rappresentata graficamente. Il grafico è di estrema utilità nei modelli economici. Sia nelle applicazioni teoriche, ossia nei modelli concettuali, che nelle analisi empiriche, è

[4] Sebbene *f (2)* si possa calcolare scrivendo semplicemente *>>>funzioneTEST(2)* sulla '*Python shell*', il testo che ho usato specifica di evitare linguaggi 'impliciti' e pertanto suggerisce di mettere in chiaro tutti i comandi che diamo al fine di una rilettura meno faticosa. Oltretutto è la modalità corretta al fine di scriverlo in foglio di testo.

assai probabile infatti ipotizzare relazioni rappresentabili graficamente tra due insiemi.

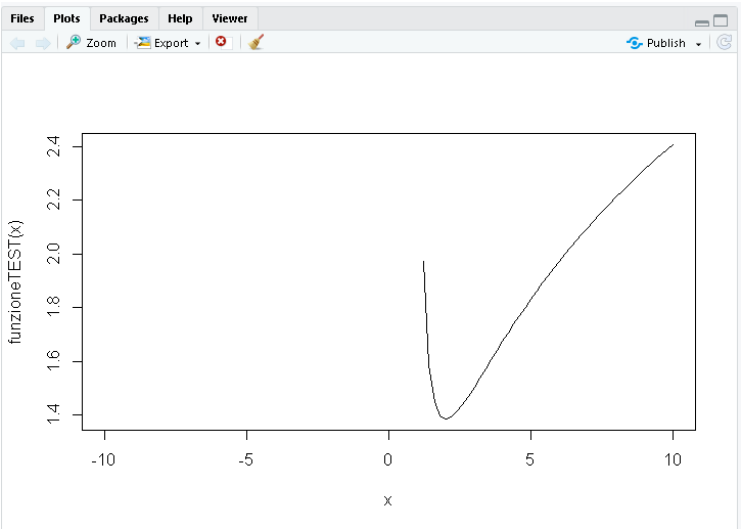
Formalizziamo il tratteggio del grafico di una funzione attraverso i tre linguaggi scelti.

Prendiamo come esempio la funzione $f(x) = ln(\frac{x^2}{x-1})$.

L'input per ottenere il grafico di una funzione su ***R***, dopo averla definita, è il seguente:

```
funzioneTEST <- function(x) log((x^2)/(x-1))
curve(funzioneTEST(x), from = -10, to = 10, n=101)
```

[5]



Il comando `curve` permette di disegnare una funzione dopo aver definito l'intervallo della variabile indipendente. Nel nostro esempio abbiamo scelto un intervallo che va da $x_0 = -10$ a $x_n = 10$ con $n = 101$. In altre parole abbiamo scelto il numero di valori assegnati alla variabile indipendente, all'interno

[5] Il plugin sul foglio di testo scrive automaticamente *log* con prima lettera maiuscola. In realtà esso è minuscolo sull'editor di R.

dell'intervallo da noi individuato, in corrispondenza dei quali far disegnare il grafico. Il sistema segnala automaticamente la presenza di non-numeri '*NaN*', ossia di punti al di fuori del dominio. In tal senso, possiamo dire che il sistema definisce il grafico senza necessità di richiedere o inserire manualmente il dominio.

Per quanto riguarda ***Octave*** il meccanismo è simile ma con alcune piccole differenze.

Definiamo prima la funzione:

```
funzioneTEST = @(x) log((x.^2)./(x-1))
```

Come si può notare, questa funzione è stata definita in modo diverso rispetto all'esempio introduttivo sulle funzioni. Precisamente osserviamo che anziché scrivere

```
log((x.^2)/(x-1))
```

abbiamo inserito anche un punto davanti al segno di divisione.

Questo è dovuto alla necessità di *vettorializzare* la funzione.

Il processo di vettorializzazione in ***Octave*** permette alla funzione di accettare come input, attraverso un solo comando, un insieme finito di valori assegnati alla variabile indipendente x. La vettorializzazione avviene nel momento di definizione della funzione inserendo un punto davanti ai comandi di moltiplicazione, divisione ed elevamento a potenza. Tale accortezza è necessaria non solo per la rappresentazione grafica ma anche per i passaggi successivi.

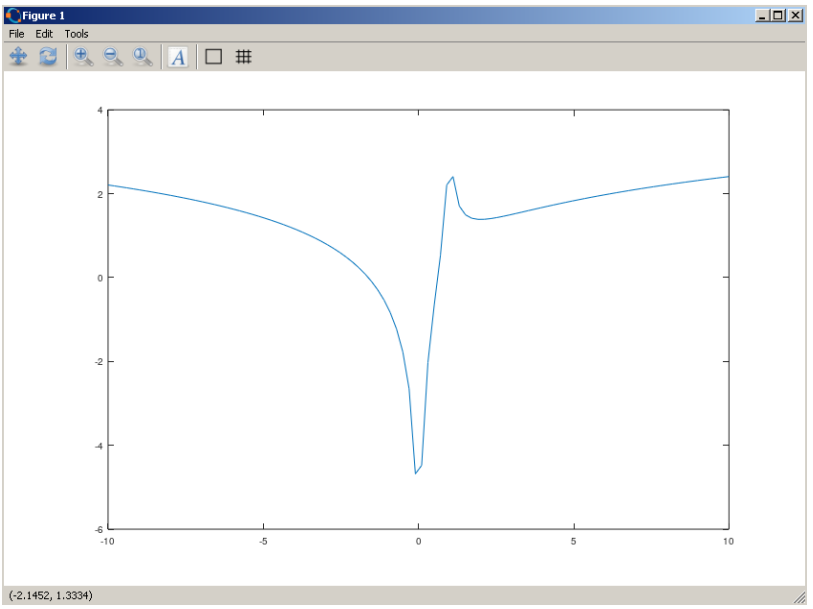
Una volta chiarito il concetto di vettorializzazione è immediata l'applicabilità circa il grafico di una funzione.

Il risvolto pratico di questa informazione potrebbe consistere nel calcolare la funzione in ogni punto di un dominio da noi scelto e formalizzato in un vettore, generando così un 'vettore codominio' di uguale lunghezza.

Da questa intuizione il primo tentativo di individuare il grafico.

```
intervallo0=linspace(-10,10,100);
% definiamo un intervallo di x dove disegnare la funzione;
%ipotizziamo un vettore di 100 punti equidistanti tra x=-10
e % x=10

plot(intervallo0, funzioneTEST(intervallo0))
%codice del tipo plot(x, f(x))
```

Con questo risultato:



Un esito piuttosto bizzarro perché contrariamente ad ***R***, ***Octave*** traccia il grafico anche laddove non potrebbe, ossia al di fuori del dominio, dividendo le ordinate in un numero composto da una parte reale ed una immaginaria. A scanso di equivoci lo stesso test con una sintassi simile è stato eseguito su ***R.***

```
a = seq(from =  -10, to = 10, length.out = 10000)
#vettore 'dominio' lungo 10000 punti da -10 a 10
```

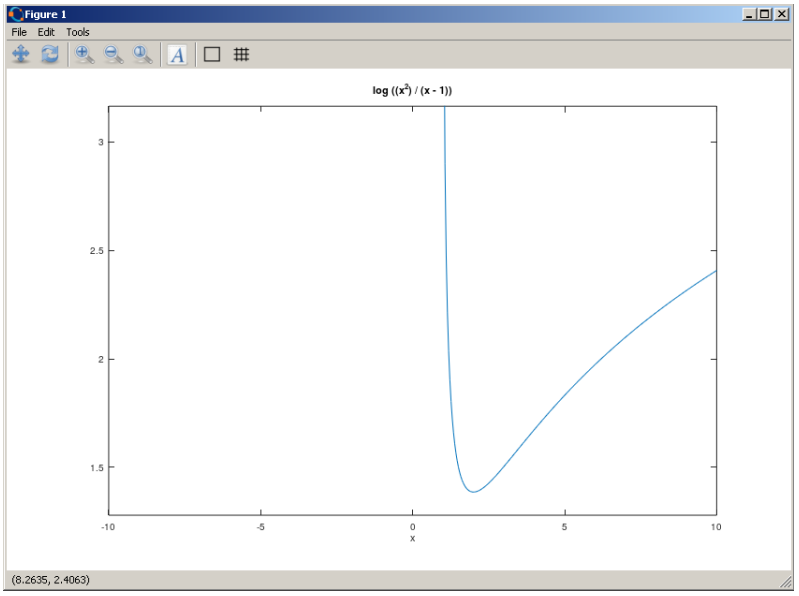
```
plot(a, funzioneTEST(a), type = "l")
#grafico i cui punti (x,y) sono congiunti da una linea.
```

Il risultato che ne emerge, anche con questa modalità, è in linea con quello individuato precedentemente sempre da ***R***.

La singolarità emersa con ***Octave*** è ovviamente risolvibile. Senza ricorrere ad impostazioni o righe particolarmente fantasiose[6], in realtà ***Octave*** possiede il codice necessario per il grafico delle funzioni. Precisamente:

```
intervallo1 = [-10 10];
ezplot(funzioneTEST, intervallo1)
```

con il seguente risultato



a patto di aver vettorializzato correttamente la funzione. Qualora così non fosse, oltre a possibili messaggi di errore potremmo anche trovarci di fronte a grafici con una sola ordinata ed ascisse tante quante ce ne sono nell'intervallo individuato.[7]

[6] Una la soluzione alternativa, ma certamente più faticosa, potrebbe consistere nell'individuazione di un vettore che esclude gli elementi immaginari attraverso operatori booleani (nel caso di Octave, 1/0),
[7] In sostanza la funzione non vettorializzata anche se ha come argomento un vettore produce uno scalare

Come abbiamo visto dall'introduzione ***Python*** è un linguaggio aperto a molteplici campi applicativi. Proprio per questo dovremmo installare diverse 'librerie' non presenti nel download generale.[8]

Importiamo i pacchetti necessari e definiamo la funzione, sia in maniera generale che in un punto:

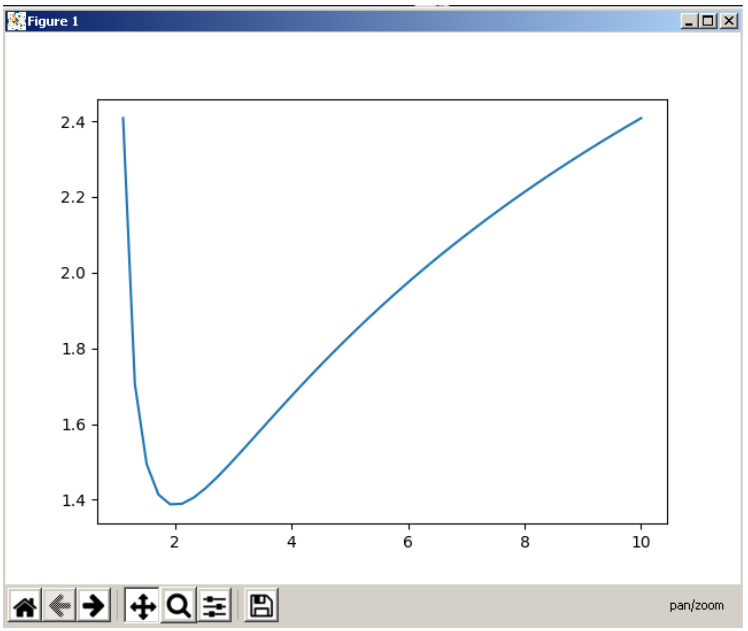
```
import math
import numpy as np
def funzioneTEST(x):
    return np.log((x**2)/(x-1))

print(funzioneTEST(2))
```

Importiamo anche il pacchetto necessario all'elaborazione grafica e provvediamo a scrivere righe di codice seguendo la stessa idea vista in ***R*** e ***Octave***:

```
import matplotlib.pyplot as plt

#definiamo un dominio al quale corrisponde uno ed un solo
#elemento del codominio attraverso la funzione

dominio=np.linspace(-10,10,100)

#vettorializziamo la funzione e definiamo il codominio.
#Possiamo anche stamparli a schermo.

vetfunzioneTEST=np.vectorize(funzioneTEST)
codominio=vetfunzioneTEST(dominio)
print(dominio)
print(codominio)

#disegniamo la funzione e chiediamo di mostrarla con il
```

[8] Il pacchetto 'math' visto nella definizione di una funzione su Python è uno di quelli già presenti all'interno della directory al momento dell'installazione. Esso non è sufficiente ai calcoli necessari ed è utile per delle elaborazioni molto "semplici". Non approfondiremo in questa tesi come installare pacchetti dall'esterno. Sebbene possa sembrare in un primo momento 'difficile' ci sono molte videoguide su internet che facilitano di molto il processo.

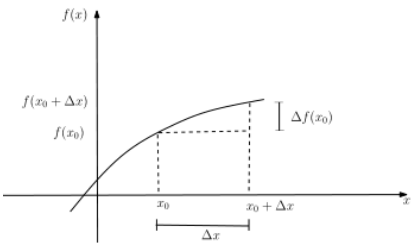
```
#pacchetto matplotlib
plt.plot(dominio, codominio)
plt.show()
```



Seppur con una sintassi a prima vista maggiormente complicata ***Python***, con l'aiuto di librerie esterne riesce a rappresentare grafici di funzioni ad una variabile.

Derivata prima e seconda di funzioni di una variabile

Uno dei principali obiettivi della teoria economica è studiare l'effetto della variazione di una grandezza economica su un'altra variabile. A tal fine è necessario il concetto di derivata.



Definito il rapporto incrementale di una funzione come il rapporto tra la variazione di ordinate e la variazione di ascisse a partire da un incrementoΔx, ossia

$$\frac{\Delta y}{\Delta x} := \frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}$$

la derivata si definisce come il limite del rapporto incrementale della funzione nel punto al tendere dell'incremento a zero $lim_{\Delta x \to 0} \frac{f(x_0+\Delta x)-f(x_0)}{\Delta x} = f'(x_0)$.

Tale punto $f'(x_0)$ esprime geometricamente (e anche numericamente) il coefficiente angolare della retta tangente alla funzione nel punto $(x_0, f(x_0))$. Un'informazione importante nell'analisi in quanto ci informa sull'andamento della funzione.

Per funzioni di una sola variabile[9] $y = f(x)$, la derivata $f'(x_0)$ misura (in modo approssimato) la variazione di $y$conseguente ad una variazione di Δx:

$$\Delta y \approx f'(x_0)\Delta x$$

Posto $\Delta x = 1$, $f'(x_0)$ misura pertanto la variazione di y dovuta all'aumento di una unità di x.

Compresa l'importanza di tale strumento, come si comporteranno i linguaggi di programmazione scelti nel calcolo della derivata?

Iniziamo da ***R***. Prima di giungere al codice che definisce la funzione derivata, procediamo per gradi. Utilizziamo quindi la *console* per approfondire ed analizzare i comandi che andranno a comporre il codice finale su ***R***.

```
> funzioneTEST
[1] function(x) log((x^2)/(x-1))
<bytecode: 0x000000000b1eb9e0>

> body(funzioneTEST)
```

[9] Ma in modo leggermente diverso anche per funzioni con più variabili

```
[1] log((x^2)/(x - 1))

> D(body(funzioneTEST), “x”)
[1] (2 * x/(x - 1) - (x^2)/(x - 1)^2)/((x^2)/(x - 1))
```

L’ultimo passaggio consiste nel calcolo vero e proprio della funzione derivata. Come possiamo vedere non è in forma semplificata e, qualora dovessimo scriverla su un foglio di carta, apparirebbe in questa forma:

$$\frac{\frac{2x}{x-1}-\frac{x^2}{(x-1)^2}}{\frac{x^2}{x-1}}.$$

Se dovessimo definire la funzione derivata con la riga sopra, tuttavia, staremmo facendo un errore. Infatti il codice appena descritto non è sufficiente a calcolare la derivata nel punto. Se chiedessimo di calcolare $f'(3)$

```
> derivatafinta <- function(x) (D(body(funzioneTEST),
"x"));
> derivatafinta(3)
[1] (2 * x/(x - 1) - (x^2)/(x - 1)^2)/((x^2)/(x - 1))
```

non otterremmo altro che l’espressione della funzione, sebbene sia stata inserita un’ascissa specifica.

Per risolvere il problema è necessario utilizzare il comando `eval`[10] . Tale comando serve a collegare la nuova *`function(x)`* con l’espressione della neonata derivata; permette inoltre anche valutazioni nel punto.

```
derivataTEST <- function(x) eval(D(body(funzioneTEST),
"x"));
#questo comando serve per assegnare al nome derivataTEST la
#funzioneTEST derivata e calcolarla nel punto.
> derivataTEST(3)
```

[10] Eval come abbreviazione di evaluate

```
[1] 0.1666667
```

Come abbiamo anticipato, la derivata è un concetto fondamentale in analisi matematica. Essa è anche ampiamente utilizzata nelle applicazioni economiche. Basti pensare a tutti quei modelli in cui si è interessati ad ottimizzare le funzioni (costi minimi, utilità massima e via dicendo); è noto che per risolvere problemi di ottimizzazione si fa uso di condizioni che coinvolgono la derivata.

Per quanto riguarda la ricerca di massimi e minimi ***R*** permette di adoperare una funzione molto utile denominata '*optimize*'.

```
optimize(funzioneTEST, interval = c(-999, 999))
```

Questa comando calcola automaticamente le coordinate del punto di minimo della funzione di partenza nell'intervallo scelto da noi, Potremmo chiedergli anche di calcolare il massimo scrivendo invece:

```
optimize(funzioneTEST, maximum = TRUE, interval = c(-999,
999))
```

Quanto abbiamo visto lo possiamo applicare anche alla derivata seconda, utile per lo studio della concavità e della convessità della funzione.

```
derivata2TEST<- function(x) eval(D(D(body(funzioneTEST),
"x"), "x"));
derivata2TEST(3) #la derivata seconda in un punto x0
curve(derivata2TEST, -10, 10) #disegno derivata seconda

#derivata seconda in forma estesa
(D(D(body(funzioneTEST), "x"), "x"))
```

Per quanto riguarda gli ultimi due linguaggi rimanenti, ***Octave*** e ***Python***, per entrambi è necessario introdurre il *calcolo simbolico*.

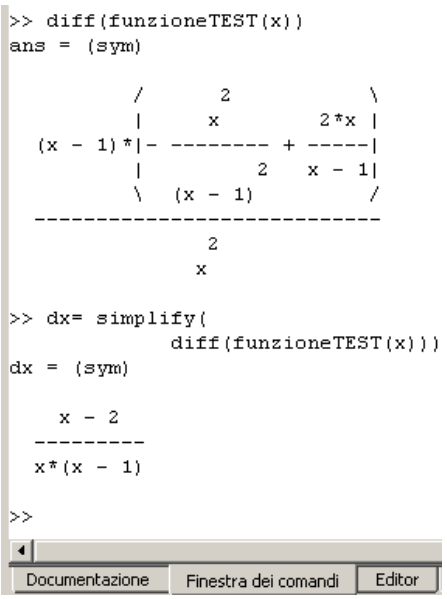
Il calcolo simbolico consiste nello svolgimento di operazioni con espressioni che contengano le variabili e non siano solo numeriche. Lo fa ***R*** automaticamente in alcuni passaggi mentre è necessario dichiararlo all'inizio del proprio calcolo sia su ***Octave*** che su ***Python***. Approfondiamo il concetto con il calcolo della derivata.

Per quanto riguarda **Octave** carichiamo il pacchetto[11] adatto a fare calcoli con le espressioni contenenti le variabili:

```
pkg load symbolic
```

Denominiamo pertanto un nuovo tipo di variabile appartenente alla classe[12] '*sym*' :

```
x=sym('x','real')
```



Completato questo passaggio è possibile calcolare la derivata in forma simbolica e, per non avere un risultato simile a quanto visto con ***R***, si può anche chiedere una semplificazione dell'espressione:

```
diff(funzioneTEST(x)) %derivata
prima
% semplificazione e assegnazione della derivata prima
dx= simplify(diff(funzioneTEST(x)))
```

[11] Alcune fonti affermano che sia necessario aver installato anche Python per far funzionare questo pacchetto. Altri affermano che serva sia Python che una libreria ad esso collegata (e presso di esso installata) denominata Sympy. Altri ancora usano un altro comando per installare inizialmente questo pacchetto. Ho condotto due prove su due PC (entrambi con Python installato di cui uno senza Sympy) e il comando inserito nel testo sembra funzionare.Informazioni sulle funzioni ed i comandi del pacchetto di Octave. .

[12] Digitando *whos* sulla finestra dei comandi è possibile vedere la tipologia di tutte le variabili che si stanno utilizzando. In alternativa appare sulla finestra a sinistra.

Possiamo notare fin da subito come ***Octave*** abbia una impostazione ed una rappresentazione simbolica molto pulita e funzionale.

Notiamo inoltre che la versione non semplificata della derivata prima è esattamente uguale a quella calcolata prima con ***R.*** In sintesi possiamo affermare che entrambi quindi supportano il calcolo simbolico, sebbene con sfumature diverse.[13]

Terminiamo quindi il nostro approfondimento su questo linguaggio di programmazione calcolando gli zeri della derivata, ossia i punti in cui si annulla.

Ricordiamo che, ai fini della ricerca di un punto di massimo o di minimo, è Condizione Necessaria, per il *Teorema di Fermat*, l'annullamento della derivata prima nel punto stesso.

```
>>solve(dx)
ans = (sym) 2
```

$f'(x)$si annulla pertanto in $x_0 = 2$. Si esegue lo stesso procedimento per la derivata seconda.

```
dx2=simplify(
             diff(dx))
solve(dx2)
```

Per quanto riguarda ***Python*** il procedimento è simile ai linguaggi visti in precedenza.

Importiamo il pacchetto per il calcolo simbolico e definiamo le variabili.

```
import sympy as sy
```

[13] L'alta diffusione di R certamente ha ampliato la platea di appassionati, studiosi e lavoratori che usano tale linguaggio di programmazione. Tra le conseguenze vi è un numero elevato di pacchetti scaricabili e tra questi è molto probabile la presenza di pacchetti per l'ampliamento del calcolo simbolico.

```
x=sy.Symbol('x')
```

Al fine di non rilevare errori[14] riscriviamo la relazione facendo provenire la funzione log dal pacchetto *Sympy.*

```
def funzioneTESTderivabile(x):
    return sy.log((x**2)/(x-1))
```

per poi derivarla con dei comandi analoghi a quelli visti in ***Octave***, ossia:

```
#assegnazione a dx1 della semplificazione della derivata
prima
dx1=sy.simplify(
    sy.diff(funzioneTESTderivabile(x), x))
#dx1 stampata a schermo
sy.pprint(dx1)
```

otterremo anche su Python la versione semplificata, e, attraverso il comando *sympy.pprint()*[15] anche in una raffigurazione semi-grafica, calcolata precedentemente da ***Octave*** ossia

```
  x - 2
---------
x*(x - 1)
```

Qualora volessimo calcolare la derivata in un punto come $x_0 = 3$scriviamo

```
print(dx1.subs(x, 3))
```

mentre l'individuazione degli zeri della derivata prima (ma potenzialmente di qualsiasi funzione) avviene tramite:

```
print(sy.solve(dx1, x))
```

[14] Nel momento in cui sto scrivendo la tesi non sono riuscito a trovare un rimedio alla necessità di ripetere - con modifiche - la definizione della funzione da analizzare. Sebbene possa sembrare un fastidio di poco conto riscrivere la funzione (seppur in questo caso cambiando poche lettere) in realtà mina alla base lo spirito con il quale sono pensati gli script risolutivi di problemi matematici, ossia essere snelli, veloci e utilizzabili su ogni funzione con pochissime modifiche. Ciò che sembra non funzionare nel nostro caso è la compatibilità tra 'numpy.log' e 'sympy.log' semplificati rispettivamente da me come 'np.log' e 'sy.log'.
[15] Questo comando - semplificato in *sy.pprint()* - stampa a schermo la funzione in una forma 'gradevole' e non su una sola riga. Maggiori informazioni alla documentazione ufficiale.

Terminiamo facendo lo stesso procedimento per la derivata seconda.

```
dx2=sy.simplify(
    sy.diff(
        sy.diff(funzioneTESTderivabile(x), x)))

sy.pprint(dx2)
print(dx2.subs(x, 2)) #ipotizziamo calcolo derivata in x=2
print(sy.solve(dx2, x))
```

Funzioni di due variabili

Definizione e grafici

La complessità dei fenomeni economici, produttivi, industriali e scientifici rende necessario l'uso di funzioni più complesse di quelle affrontate nel capitolo precedente. Ampliamo pertanto la nostra trattazione includendo funzioni di due variabili indipendenti; precisamente $f: \mathbb{R}^2 \to \mathbb{R}$ definita da $f(x, y) = 4x^3 - 3xy + 3y^2 + 9$.

Per rappresentare mediante il grafico funzioni da $\mathbb{R}^2$ a $\mathbb{R}$ sono necessarie tre dimensioni. In quanto riporta il problema da tridimensionale a bidimensionale, una rappresentazione geometrica utilizzata in alternativa per le funzioni di due variabili è quella mediante le curve di livello della funzione.

In sintesi dato (x, y) calcoliamo $f(x, y)$ e ne indichiamo il valore con z. Tracciamo quindi sul piano xy il luogo delle punti (x, y) in corrispondenza delle quali f assume lo stesso valore z. Esso è detto insieme di livello di f o, nel caso di funzione di due variabili, *curva di livello*.

Tale metodo è utilizzato anche nella microeconomia per individuare, nell'ambito della produzione, gli insiemi di *isocosti* e di *isoquanti*, ossia quell'insieme di curve tali da fornire diversi livelli di costi o di produzione con diverse combinazioni di fattori produttivi.

Considerata la potenza di calcolo e la possibilità di affidare ad una macchina la rappresentazione 3D, possiamo provare a rappresentare graficamente sia le curve nel piano che la nostra funzione $f: \mathbb{R}^2 \to \mathbb{R}$ in tre dimensioni.

Definiamo su ***R*** una funzione con due variabili indipendenti e scriviamo le righe di codice necessarie alle diverse rappresentazioni geometriche.

```
#definisco la funzione
funzionetesi <- function(x,y) 4*x^3+3*x*y+3*y^2+9;
#definisco due vettori uguali x e y
x <- y <- seq(from = -10, to = 10, length.out = 50)
#creiamo una matrice i cui elementi sono le combinazioni
f(x,y)
z = outer(x,y,funzionetesi);
```

Per questo ultimo passaggio è necessario fare un ulteriore approfondimento.

Data $f(x, y)$ e due *array*[16] $\bar{x}$ e $\bar{y}$, il comando `outer` crea una matrice con al suo interno le diverse combinazioni risultanti da $f(\bar{x}, \bar{y})$. In altre parole, attingendo dalla documentazione, *"`outer` takes two vectors and a function (that itself takes two arguments) and builds a matrix by calling the given function for each combination of the elements in the two vectors"*. Facendo un esempio, data una

[16] Termine che, nell'ambito dei linguaggi di programmazione, indica una struttura di dati formata da più componenti ordinate dello stesso tipo (per esempio, numeri interi o caratteri o variabili booleane) e identificate ognuna da uno o più indici che ne determinano univocamente la posizione nell'insieme. Un array può essere formato da una sola riga (con n componenti) e allora ognuno dei suoi elementi è identificato da un indice che ne determina univocamente la posizione: in tale caso realizza nel linguaggio di programmazione il corrispondente di un vettore n-dimensionale in matematica.Da Treccani- Enciclopedia della matematica

funzione $f(x,y) = x^2 + y^2$ e due vettori identici $\bar{x} = \bar{y} = (1;2)$, il comando `outer` esprimerà la matrice

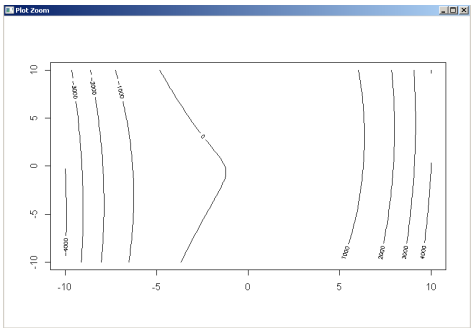
$$\begin{pmatrix} f(1,1) & f(1,2) \\ f(2,1) & f(2,2) \end{pmatrix} \rightarrow \begin{pmatrix} f(x_1,y_1) & f(x_1,y_2) \\ f(x_2,y_1) & f(x_2,y_2) \end{pmatrix}$$

e quindi in forma generica il comando `outer` calcola una matrice $n \times m$ del tipo

$$\begin{pmatrix} f(x_{1,}y_1) & \cdots & f(x_1,y_m) \\ \vdots & \ddots & \vdots \\ f(x_n,y_1) & \cdots & f(x_n,y_m) \end{pmatrix}$$

Tale comando permette quindi di calcolare una matrice simile a quella sopra, necessaria per disegnare le curve di livello, il grafico tridimensionale ed una ulteriore analisi grafica dell'andamento della funzione. Procediamo quindi con il codice:

```
#curve di livello
curvelivello=contour(x,y,z)
#grafico tre dimensioni
persp(x,y,z, theta=-30,phi=15,ticktype="detailed")
#crescita e decrescita della funzione
image(x,y,z)
```



Curve di livello $f(x,y) = z$con ***R***.

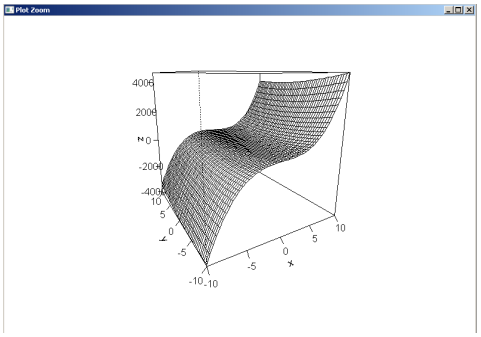
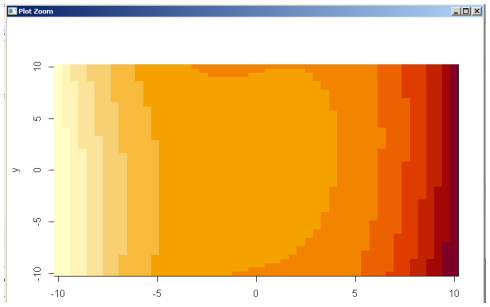


Grafico tridimensionale; modificando nel codice le cifre di `theta` e `phi` si avranno differenti prospettive.



`image(...)` evidenzia l'andamento dell'altezza della funzione. Da chiaro a scuro implica un aumento del valore della funzione $f(x, y)$.

Per quanto riguarda ***Octave*** delineiamo un processo simile. Dopo le definizioni introduttive

```
funzionetesi = @(x,y) (4.*x.^3+3.*x.*y+3.*y.^2+9)
x=y=linspace(-10,10,50)
```

al fine di poter disegnare le curve di livello è necessario creare la matrice risultante dalle diverse combinazioni di x e y.

In questo caso il procedimento è meno intuitivo di ***R*** e la matrice Z si definisce in due passaggi. Ricordando che ***Octave***, come come gli altri linguaggi, è *case sensitive*[17], calcoliamo prima due matrici separate attraverso il comando:

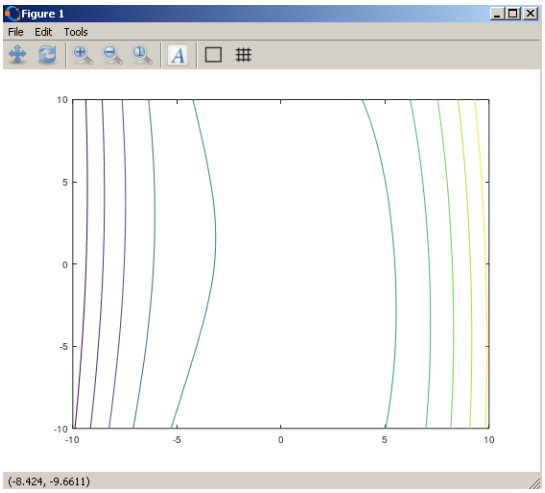
```
[X,Y] = meshgrid(x,y);
```

Tali matrici `X` e `Y` serviranno da input nel processo combinatorio utile per le curve di livello.[18] Infatti:

[17] Tengono conto in maniera diversa di maiuscole e minuscole
[18] Possiamo anche vedere la struttura di tali matrici digitando nella console rispettivamente `X` e `Y`

```
Z = funzionetesi(X,Y)
contour(X,Y,Z)
```

con la conseguente rappresentazione grafica

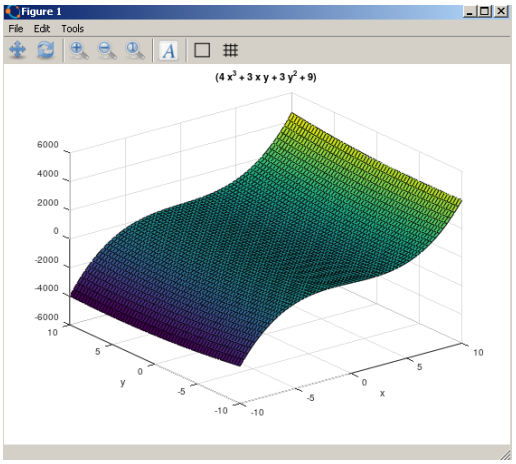


Terminiamo il nostro passaggio su ***Octave*** attraverso la rappresentazione grafica della nostra funzione.

```
ezsurf(funzionetesi, dom=[-10,10])
```

o in alternativa inserendo manualmente le combinazioni viste prima

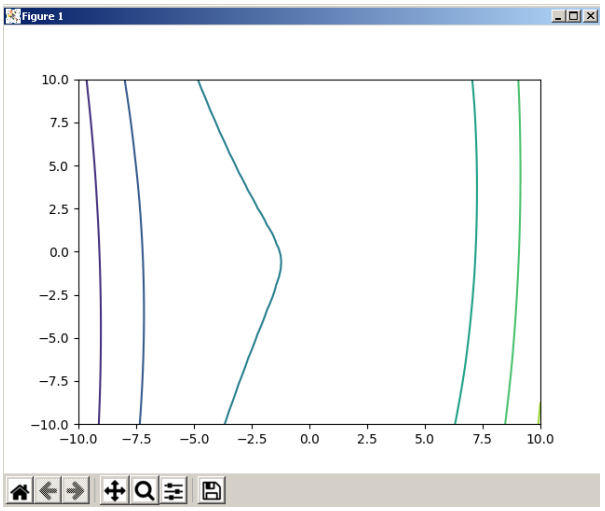
```
surf(X,Y,Z)
```



Ultimiamo la trattazione grafica attraverso ***Python.***

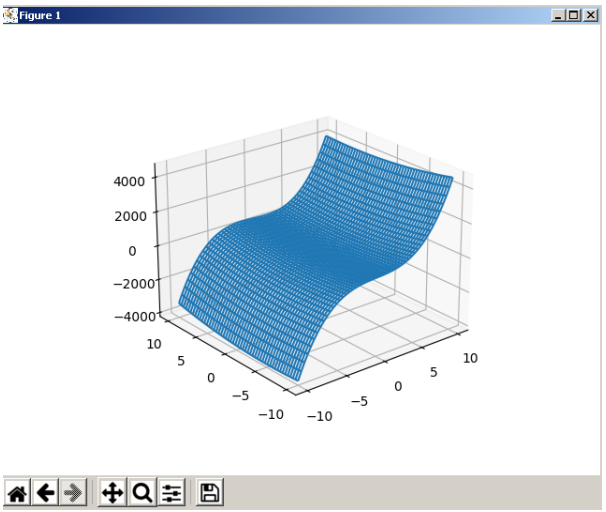
La raffigurazione delle curve di livello è simile a quanto visto con ***Octave*** grazie al modulo *pyplot* del package *Matplotlib* compatibile con il modulo *Numpy*.

```
import matplotlib.pyplot as plt
import numpy as np

def funzionetesi(x,y):
    return 4*x**3 - 3*x*y + 3*y**2 +9

x=y=np.linspace(-10,10,50)
X,Y=np.meshgrid(x,y)
Z=funzionetesi(X,Y)
plt.contour(X,Y,Z)
plt.show()
```



Possiamo continuare nella rappresentazione tridimensionale sempre attraverso il modulo *matplotlib.pyplot*

```
#creiamo una nuova figura tridimensionale
fig = plt.figure()
#creiamo gli assi a 3 dimensioni e creiamo il grafico
usando #il reticolo adoperato per le curve di livello
plt.axes(projection="3d").plot_wireframe(X, Y, Z)
plt.show()
```



A differenza degli altri linguaggi in questo caso è possibile muovere la figura liberamente usando il puntatore. Possiamo rappresentare diverse elaborazioni grafiche, grazie a diversi comandi desumibili dalla letteratura[19].

Un problema di ottimizzazione vincolata

Affronteremo ora problemi di ottimizzazione di funzioni di due variabili. Nel dettaglio, analizzeremo un esempio di ottimizzazione vincolata[20]. Tale quesito consiste nella ricerca dei punti stazionari di una funzione che soddisfino simultaneamente altri vincoli espressi da equazioni o diseguaglianze.

Prenderemo come esempio un modello tipico dell'analisi microeconomica: la *teoria del consumatore*.

In questo modello vi è un individuo che si affaccia sul mercato con preferenze ben definite. Assumendo i prezzi come dati, il consumatore si preoccupa soltanto di

[19] https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html

[20] L'ottimizzazione vincolata si differenzia dall'ottimizzazione libera in quanto, quest'ultima, consiste nella ricerca di tutti i punti critici nel dominio, senza particolari vincoli.

allocare il proprio reddito, comprando i beni in vendita, in modo tale da soddisfare al meglio le proprie preferenze.

L'ipotesi di preferenze ben definite incorpora l'idea che l'utilità derivante dalla scelta del paniere di beni sia in qualche modo 'misurabile'. Non stiamo soltanto affermando che il soggetto preferisca un paniere ad un altro, bensì stiamo esprimendo tale preferenza mediante un fattore numerico che rappresenti l'utilità derivante dalla scelta del paniere di beni comprato. Da ciò deriva la definizione delle preferenze del consumatore attraverso una funzione di utilità $U\ (x_1, x_2,..., x_n)$, le cui variabili sono le quantità scelte di $n$beni che costituiscono il paniere.

Ciò detto, è necessario altresì affermare che non ci interessano particolarmente le proprietà cardinali (quindi i valori esatti) della funzione di utilità, quanto invece quelle ordinali (legate all'ordine), utili per studiare le preferenze.

Una funzione matematica molto utilizzata per descrivere le preferenze è la funzione Cobb-Douglas; questa funzione di utilità, nel caso di un paniere con due beni, si può scrivere nella forma$U(x_1, x_2) = A\ x^{\alpha} y^{\beta}$ con A, αe βparametri positivi.

Il consumatore inoltre, date le sue preferenze ed i prezzi, deve considerare anche un'altra questione: il proprio vincolo di bilancio.

Tale vincolo esprime la quantità di denaro che il consumatore è disposto a spendere. Possiamo distinguere vincoli di disuguaglianza, dove il consumatore è pronto a spendere denaro in maniera inferiore o uguale ad un massimo, e vincoli di uguaglianza, dove è espressa la cifra esatta che egli vuole spendere.

Distinguiamo pertanto, con x e y le quantità dei due beni, P_i il prezzo del bene x, P_2 il prezzo del bene y ed infine W il reddito disponibile:

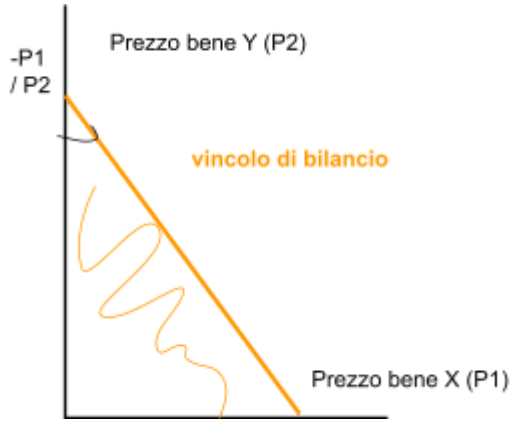
$$\textit{Vincolo di uguaglianza} \rightarrow P_1 x + P_2 y = w$$

$$\textit{Vincolo di disuguaglianza} \rightarrow P_1 x + P_2\, y \leq w$$

Un problema di massimizzazione dell'utilità con un vincolo di bilancio lo possiamo anche raffigurare graficamente.

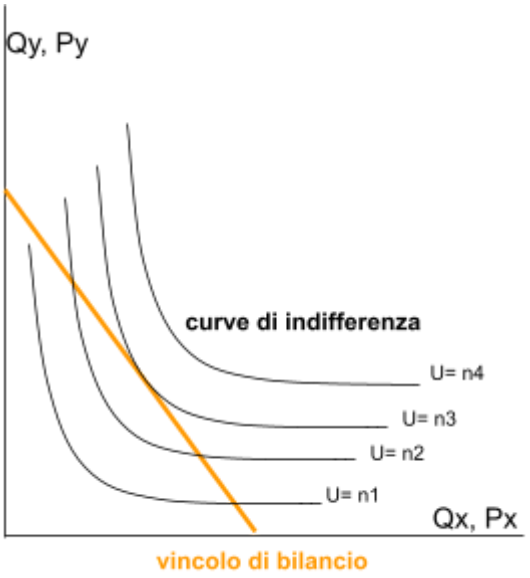
Consideriamo una funzione di utilità $U(x_1, x_2) = A\, x^{\alpha} y^{\beta}$ con A, α e β parametri positivi ed un vincolo di uguaglianza $P_1 x + P_2\, y = w$.

Il vincolo di bilancio è una retta con pendenza negativa $-\frac{P_1}{P_2}$. Tale retta indica tutte quelle combinazioni di beni del paniere che il consumatore è in grado di comprare. Qualora ipotizzassimo un vincolo di disuguaglianza, affermeremmo implicitamente la disponibilità del consumatore a spendere *anche* una cifra minore rispetto a quella massima a sua disposizione. Con questa ipotesi tra i panieri ammissibili rientrerebbero anche le combinazioni di beni tra gli assi cartesiani e la retta.



La funzione di utilità è rappresentabile graficamente ricorrendo al concetto di curve di livello.

Data una funzione$U(x, y) = A\, x^{\alpha} y^{\beta}$,con A, α e β fissati, assegniamo dei valori maggiori di zero ad U, così da disegnare, per ogni valore fissato, una differente curva di livello. Tali curve, nel caso di funzioni di utilità, si chiamano anche *curve di indifferenza.*



Le curve di indifferenza rappresentano le combinazioni di beni tali da generare un uguale livello di utilità. Con utilità marginali positive, più la curva di indifferenza è in alto e più l'utilità totale sarà maggiore.

La medesima curva di indifferenza indica lo stesso livello di utilità ma con diverse combinazioni di x ed y. Il punto di massimo, nel caso di vincolo di uguaglianza, consisterà nella scelta della quantità del bene x ed y tale da essere tangente alla retta del vincolo di bilancio. In quel punto infatti si avrà massimizzato la propria utilità soddisfacendo il vincolo di bilancio[21]. Inoltre, nel punto ottimale, si eguagliano le

[21] Con ipotesi classiche il vincolo si definisce 'attivo', Da un punto di vista geometrico, nel punto di equilibrio, le pendenze saranno le medesime.

pendenza della retta $-\frac{P_1}{P_2}$ e della curva di indifferenza, laddove quest'ultima, grazie al teorema della funzione implicita, equivale al rapporto tra le derivate parziali nel punto $-\frac{\frac{\partial U}{\partial x}(x_0,y_0)}{\frac{\partial U}{\partial y}(x_0,y_0)}$. Tale rapporto rappresenta il *saggio marginale di sostituzione*; esso misura la quantità aggiuntiva del bene *y* necessaria a compensare la perdita di una unità del bene *x* al fine di mantenere invariato il livello di soddisfazione. Diverso invece è il significato della pendenza del vincolo di bilancio: essa indica la quantità del bene sull'asse delle ordinate al quale il consumatore deve rinunciare per poter acquistare un'unità aggiuntiva del bene sull'asse delle ascisse.

Dopo aver introdotto la teoria del consumatore possiamo scrivere uno script in ***Python*** per massimizzare una funzione di utilità dato un vincolo di uguaglianza.

Consideriamo il seguente problema:

$$\text{massimizzare } f\,(x,y) = x\,y$$

$$\text{con il vincolo } h(x,y) \equiv x + 4\,y = 16$$

Formalizziamo la funzione di utilità e l'equazione del vincolo:

```
import matplotlib.pyplot as plt
import sympy as sy

#prepariamo il calcolo simbolico
x = sy.Symbol('x')
y = sy.Symbol('y')

#funzione di utilità
alfa = 1
beta = 1
```

```
A = 1
def utilita(x,y):
    return A * x**alfa * y**beta
print('funzione di utilità:', utilita(x,y))

#scriviamo l'equazione della retta del vincolo di bilancio
P1=1
P2=4
w=16
def vincolo(x,y):
    return P1*x + P2*y - w
print('il nostro vincolo:', vincolo(x,y))
```

Ricordando il Teorema dei moltiplicatori di Lagrange che individua le Condizioni Necessarie per l'esistenza di un punto di massimo vincolato, risolviamo il sistema composto da tre equazioni

$$\{\frac{\partial L}{\partial x} = 0; \frac{\partial L}{\partial y} = 0; \frac{\partial L}{\partial u} = 0\}$$

ottenuto annullando il gradiente della funzione Lagrangiana

$$L(x, y, u) \equiv f(x, y) - u[h(x, y) - c]$$

```
#funzione Lagrangiana
u = sy.Symbol('u')
def L(x,y,u):
    return utilita(x,y) -
u*(vincolo(x,y))
print('la nostra funzione
lagrangiana', L(x,y,u))

#derivate parziali funzione
lagrangiana
Lx = sy.simplify(
    sy.diff(L(x,y,u), x))
Ly = sy.simplify(
    sy.diff(L(x,y,u), y))
Lu = sy.simplify(
```

```
    sy.diff(L(x,y,u), u))

print('Le derivate parziali
di L(x,y,u)')
print('Lx=', Lx)
print('Ly=', Ly)
print('Lu=', Lu)
```

Risolviamo il sistema tra le derivate parziali ed esplicitiamo l'unico candidato come soluzione al problema di massimizzazione:

```
#risolviamo il sistema
puntistazionari=sy.solve((Lx, Ly, Lu), x, y, u, dict=True,
set=False)
print('ecco le soluzioni del sistema di derivate parziali')
print('soluzioni=', puntistazionari)
```

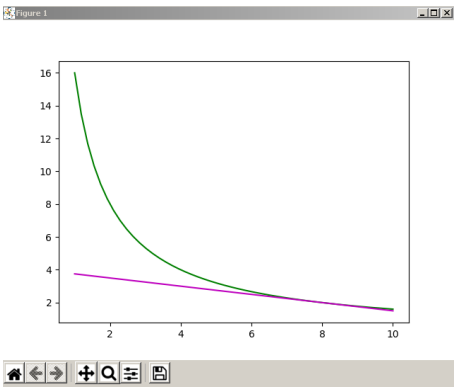
Il nostro script ci indica la seguente soluzione:

```
soluzioni= [{x: 8, y: 2, u: 2}]
```

Con poche righe di codice aggiuntive è possibile rappresentare la condizione di tangenza tra la curva di livello ed il vincolo di uguaglianza:

```
#definiamo le curve da disegnare
def utilitacurva(x):
    return (w/(x**alfa))**(1/beta)
def vincoloretta(x):
    return (-P1*x)/Pb + w/P2

#grafico della curva di livello
dominio0=np.linspace(1,10,50)
vetutilita=np.vectorize(utilitacurva)
codominio0=vetutilita(dominio0)
plt.plot(dominio0, codominio0, 'g')

#grafico del vincolo
vetvincolo=np.vectorize(vincoloretta)
```

```
codominio1=vetvincolo(dominio0)
plt.plot(dominio0, codominio1, 'm')

plt.show(block=True)
```



Possiamo ottenere le medesime soluzioni al problema di ottimizzazione anche su

Octave:

```
% prepariamo calcolo simbolico
pkg load symbolic
x=sym('x','real');
y=sym('y','real');
%funzione di utilità
alfa = 1
beta = 1
utilita=@(x,y) x.^alfa .* y.^beta
%equazione vincolo di bilancio
P1=1
P2=4
w=16
vincolo=@(x,y) P1*x + P2*y - w
%equazione della retta
vincoloretta=@(x)(-P1*x)/P2 + w/P2
%funzione lagrangiana
u=sym('u','real');
L=@(x,y,u) utilita(x,y)-u*vincolo(x,y)

Lx=simplify(diff(L(x,y,u),x))
Ly=simplify(diff(L(x,y,u),y))
Lu=simplify(diff(L(x,y,u),u))
```

```
%risolviamo il sistema di cui sopra ed otteniamo le
soluzioni
%con il seguente comando
[xs, ys, us]= solve(Lx, Ly, Lu)
```

Anche con ***R*** è possibile raggiungere risultati simili, sebbene la risoluzione di tali sistemi sia affidata a pacchetti esterni piuttosto complessi.

Conclusioni

All'interno di questa tesi abbiamo utilizzato tre diversi linguaggi di programmazione per formalizzare alcuni dei principali quesiti matematici più semplici legati all'economia: calcolo del valore di una funzione in uno o più punti, rappresentazione mediante il grafico, ricerca di massimi e minimi per funzioni di una variabile ed ottimizzazioni vincolate.

Ciò che emerge è una particolare flessibilità da parte dei diversi linguaggi nell'elaborare compiti matematici, risolvere casi o adattarsi nella risoluzione di specifici problemi.

Tra i linguaggi utilizzati quello maggiormente flessibile, adattabile, semplice e potente sembra essere ***Python***.

Sebbene in un primo momento possa apparire come il più complesso, soprattutto se comparato agli altri due, nel tempo riesce a mostrare tutta la sua affidabilità e maneggevolezza. Da segnalare inoltre la sua enorme diffusione ed i vantaggi di questa popolarità: molteplici librerie disponibili, soluzioni ai problemi di sintassi più frequenti e innumerevoli guide all'utilizzo.

Importante anche ***Octave***, 'clone' di Matlab, il quale presta parte dei propri meccanismi proprio ad alcune librerie di ***Python***[22].

Al terzo posto ***R***, strumento ugualmente potente e capace di reggere il confronto con gli altri due, anche se talvolta con qualche difficoltà per l'utente (v. ottimizzazione vincolata).

In ogni caso, al di là del linguaggio scelto, la forza di calcolo dei computer e l'intuito umano, riassumibili in script riutilizzabili e sempre modificabili, apriranno le porte ad una modellistica economica sempre più profonda. Questo è il vero motivo per il quale è importante cominciare ad imparare i meccanismi della formalizzazione informatica dei problemi economici fin da subito.

Bibliografia

Roncaglia, A. 2016. *Breve storia del pensiero economico*, Bari: Laterza

Simon, C.P. and Blume, W., 2015. *Matematica per le scienze economiche*, Milano: Egea

Frank, R., Cartwright, E. Piras, R,. 2016 *Microeconomia VII edizione*, Milano: McGraw-Hill

R Foundation, 2021. *The Comprehensive R Archive Network v.4.0.2.* Wien: GNU license cran.r-project.org

Free software foundation, 2021. *GNU Octave v.6.1.0* Boston: GNU license www.octave.org

Python software foundation, 2021. *Python v.3.7.0.* Wilmington: Open source license. www.python.org

Severance, C., 2009. *Python per tutti*, Ann Arbor: Creative Commons

[22] Un osservatore attento avrà notato che spesso le formalizzazioni matematiche tra Octave e Python non sono molto diverse.

https://www.moreware.org/books/python_per_tutti.pdf

Mazzia, A. 2017. *Appunti sparsi su Octave*. Unipd, Creative Commons

https://dispense.dmsa.unipd.it/janna/tutorial_Mazzia.pdf

Frascati, F., 2008. Formulario di statistica con R. Firenze, GNU license

https://cran.r-project.org/doc/contrib/Frascati-FormularioStatisticaR.pdf