Università degli Studi
di Cagliari

Facoltà di Ingegneria Elettronica
Corso di Calcolo Numerico 2

Risoluzione di Sistemi di Equazioni non Lineari

Studente
Paolo Meloni

Docente
Giuseppe Rodrigez

Introduzione

La risoluzione di equazioni e di sistemi di equazioni non lineari e un problema di fronte al quale ci si trova spesso nell'ambito scientifico ed ingegneristico. Come diretta conseguenza di ciò si ha la necessità di saper risolvere tali classi di equazioni in maniera veloce ed efficiente.
Tale necessità si scontra però con l'oggettiva difficoltà realizzativa causata dalla non linearità delle equazioni che andremo a trattare.
Per porre rimedio a tale situazione sono stati sviluppati metodi di tipo iterativo estremamente efficienti che, nell'era dell'elettronica, ci mettono in condizione di risolvere problemi che fino a poco meno di un secolo fa risultavano essere estremamente ostici se non addirittura irrisolvibili a meno di non applicare pesanti approssimazioni.
Scopo di questa tesina è quello di analizzare tali metodi in maniera da comprenderne il funzionamento, le potenzialità e gli eventuali limiti.

Polinomi

Il problema di individuazione degli zeri di un polinomio algebrico risulta essere molto semplificato se lo si guarda da un'altra prospettiva.
Dato il generico polinomio

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

possiamo ricavarne, dividendo tutti i coefficienti per a_n, il seguente polinomio monico

$$\tilde{p}_n(x) = x^n + b_{n-1} x^{n-1} + \cdots + b_1 x + b_0$$

che avrà gli stessi zeri del polinomio di partenza.
Partendo dal polinomio monico possiamo ricavate la seguente matrice

$$C_n = \begin{bmatrix} -b_{n-1} & -b_{n-2} & \cdots & -b_1 & -b_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

detta matrice compagna ed avente autovalori coincidenti con gli zeri del polinomio $\tilde{p}_n(x)$.
Dato che possediamo algoritmi per il calcolo degli autovalori molto efficienti possiamo sfruttarli per calcolare degli zeri del polinomio dato.
Uno degli algoritmi più efficaci per il calcolo degli autovalori è quello QR. Tale algoritmo è di tipo iterativo e prevede, data la matrice A di qui si vogliono conoscere gli autovalori, di calcolare la fattorizzazione QR di tale matrice ed ottenere la matrice per il passo successivo compiendo il prodotto R*Q. Sotto la condizione che gli autovalori abbiano tutti distinti in modulo iterando il procedimento la matrice A convergerà ad una matrice triangolare superiore T, con autovalori quindi sulla diagonale, avente gli stessi autovalori di A. Nel caso invece di autovalori complessi coniugati e reali, ma con molteplicità superiore ad uno, si verranno a creare dei blocchi quadrati di dimensione pari alla molteplicità dell'autovalore. Una possibile implementazione in Matlab di quest'algoritmo può essere la seguente

```
p=!!polinomio
n = length(p)-1;
C=[-p(2:n+1)/p(1);eye(n-1) zeros(n-1,1)]
A = [0 0 0;0 0 0 ; 0 0 0];
k = 0;
while ((norm(diag(C,-1),inf) > eps) & (k<N))
   k = k+1;
   A = C;
   [Q,R] = qr(C);
   C = R*Q
end
ROOTS = diag(C);
```

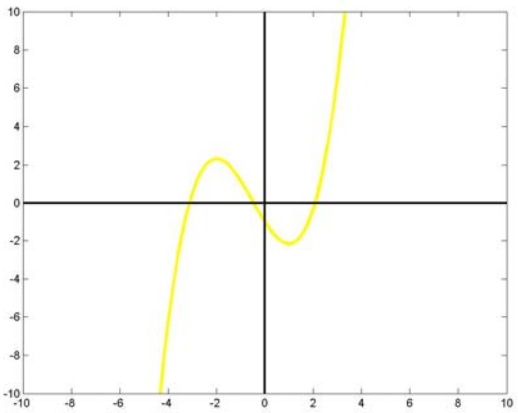
L'algoritmo, dato un polinomio, costruisce la relativa matrice compagna e poi procede con le iterazioni finché o la norma infinito del vettore costituito dagli elementi della sottodiadonale della matrice compagna, che sono gli unici elementi diversi da zero nel triangolo da zero, raggiunge dimensione inferiore della eps (precisione di macchina) oppure viene raggiunto il numero massimo di iterazioni da noi precedentemente impostato.
Andiamo ora a testare tale algoritmo sul seguente polinomio.

$$f(x)=\frac{2x^3+3x^2-12x+h}{6}$$

con h costante che andremo ad impostare volta per volta.
Iniziamo con l'impostare $h=-6$. La funzione quindi è

$$f(x)=\frac{2x^3+3x^2-12x-6}{6}$$

Il grafico della funzione è il seguente



Il nostro algoritmo giunge dopo 91 iterazioni al seguente risultato

$$C=\begin{bmatrix}-3.1446 & 5.5001 & -2.1316\\ 0 & 2.0794 & -0.4525\\ 0 & 0 & -0.4360\end{bmatrix}$$

quindi

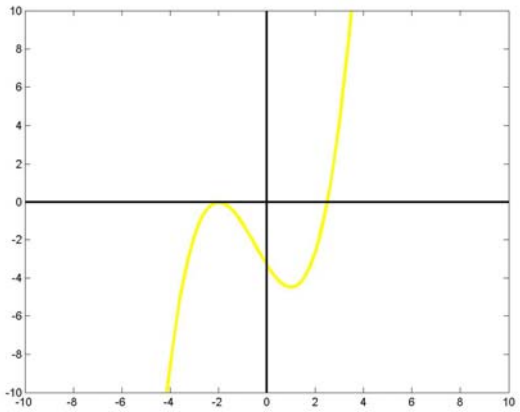
$$Roots=\left[-3.1446,2.0794,-0.436\right]$$

che è lo stesso risultato che otteniamo con la funzione di Matlab roots().

Analizziamo ora il caso che si ha per $h=-20$. La funzione sarà quindi

$$f(x)=\frac{2x^3+3x^2-12x-20}{6}$$

il cui grafico è il seguente



Si vede che si ha un polo a molteplicità doppia nel punto $x = -2$. La funzione roots() ci da i seguenti risultati

$$Roots = [2.5, -2, -2]$$

Il nostro algoritmo invece dopo 100 iterazioni, limite massimo da noi imposto, restituisce il seguente risultato

$$C = \begin{bmatrix} 2.5 & 4.1484 & 9.6908 \\ 0 & -2.0197 & -3.8572 \\ 0 & 0.001 & -1.9083 \end{bmatrix}$$

Per lo zero a molteplicità singola si ha avuto convergenza, ma che per via dei due zeri coincidenti la sottomatrice

$$C' = \begin{bmatrix} -2.0197 & -3.8572 \\ 0.001 & -1.9083 \end{bmatrix}$$

non può convergere. Se noi calcoliamo il polinomio caratteristico di tale matrice e poi ne cerchiamo gli zeri otteniamo

$$\det(sI - C') = (\lambda + 2.0197)(\lambda + 1.9083) + 0.0038752 = 0$$

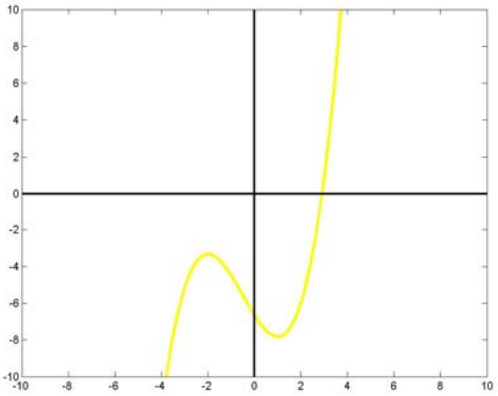
e come soluzione

$$\lambda_{1,2} \approx -2$$

dove il circa uguale deriva dagli errori d'approssimazione presenti sulla matrice.
Consideriamo ora come ultimo caso quello di $h = -40$, cui corrisponde la funzione

$$f(x) = \frac{2x^3 + 3x^2 - 12x - 40}{6}$$

Il grafico è il seguente



Si vede chiaramente che la funzione ha una sola radice reale. Dato che il polinomio è di terzo grado dovremo avere altre due radici complesse coniugate. La funzione roots() ci fornisce il seguente risultato

$$Roots = \left[2.9141, -2.2071 + 1.4114\text{i}, -2.2071 - 1.4114\text{i}\right]$$

Il nostro algoritmo invece dopo 1000 iterazioni ci da la seguente soluzione

$$C = \begin{bmatrix} 2.9141 & -7.9140 & 18.0965 \\ 0 & -2.7794 & 6.4803 \\ 0 & -0.3580 & -1.6347 \end{bmatrix}$$

Si vede che abbiamo ottenuto lo stesso comportamento del caso di radici reali coincidenti.
Se come prima prendiamo la sottomatrice

$$C' = \begin{bmatrix} -2.7794 & 6.4806 \\ -0.3580 & -1.6347 \end{bmatrix}$$

e ne calcoliamo gli autovalori con la formula analitica otteniamo

$$\lambda_{1,2} = -2.2071 \pm 1.4114\text{i}$$

che cono esattamente quelli trovati dalla funzione roots() di Matlab.
L'unico modo per far sì che il nostro algoritmo sia in grado di individuare anche le radici multiple e quelle complesse e quello di metterlo in condizione di individuare i blocchi che non si riducono, estrarli e calcolarne gli autovalori con altri metodi che riescano ad estrarli nonostante questa situazione. Questo è proprio ciò che viene fatto dalla funzione roots() di Matlab.

Metodi Monodimensionali

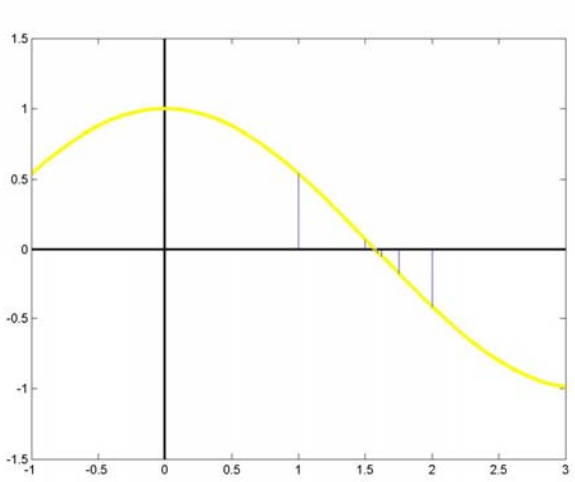
Metodo di Bisezione

Il metodo di bisezione è una procedura che dato un intervallo permette di trovare, qualora vi fosse, uno zero al suo interno.
Esso si basa sul seguente assunto: data una funzione $f(x)$ se quest'ultima ha uno zero all'interno dell'intervallo $[a,b]$ allora $f(a)$ ed $f(b)$ avranno segno opposto. Basandoci su questa proprietà possiamo andare a restringere il cerchio suddividendo l'intervallo $[a,b]$ in due sottointervalli $[a,c]$ e $[c,b]$, con c punto medio tra a e b, ed applicare la proprietà ai due sottointervalli. Una volta individuato quale dei due intervalli contiene lo zero butto via l'altro e riapplico la procedura sul nuovo intervallo ottenuto. Iteriamo questa procedura fintanto che non troviamo esattamente lo zero oppure finche non delimitiamo un intervallo talmente ristretto da far sì che assumere come zero il punto medio di tale intervallo risulti essere un'approssimazione accettabile.
Una possibile implementazione di tale metodo nel linguaggio Matlab può esser la seguente:

```
fa =!! f(x) calcolata in a
fb = !! f(x) calcolata in b
fc = 1
if  (sign(fa) ~= sign(fb))

      while ((abs(a-b) > T*eps)  & (abs(fc)> T*eps) &(N>k))
          c = (b+a)/2;
          fc = !!f(x) calcolata in c
          if ((sign(fa) ~= sign(fc)) & (sign(fb) == sign(fc)) )
             b = c;
          else ((sign(fa) == sign(fc)) & (sign(fb) ~= sign(fc)) )
              a=c
          end
          k=k+1
      end
      k
      fc
      c
      else
   error('non si ha passaggio per lo zero o si ci passa un numero pari di volte')
end
```

L'algoritmo inizia con un'istruzione if else che ha il compito, controllando i segni della funzione calcolata in *a* e *b*, se all'interno dell'intervallo è presente uno zero oppure no. In caso positivo l'algoritmo procede, altrimenti lancia un messaggio di errore che ci informa dell'assenza di zeri in tale intervallo o della presenza di un numero pari. All'interno del if si ha un costrutto while che ha il compito di iterare il procedimento fintanto che non venga trovato una buona approssimazione dello zero o che l'intervallo si riduca al di sotto di una dimensione da noi fissata pari a T volte eps, dove eps è la precisione di macchina. Dentro il while abbiamo oltre alle istruzioni per il calcolo del punto *c* e di $f(c)$ delle istruzioni if-else. Esse svolgono il compito di scegliere quale intervallo usare per l'iterazione successiva. Testeremo ora tale metodo sulla funzione $y = \cos(x)$, infetti questa funzione può mettere in evidenza tutti gli eventuali comportamenti di tale algoritmo



Ponendo

- $[a,b]=[1,3]$
- $T=10^{3}$

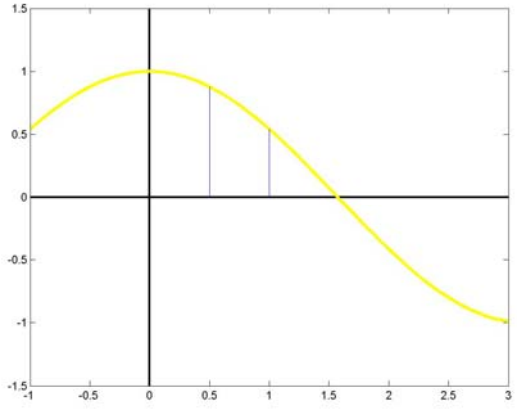
Arriviamo nel giro di 41 iterazioni al seguente risultato

- $x_{zero}=1.5708$
- $f(x_{zero})=\text{-}1.6514*10^{-13}$

L'algoritmo ha terminato di iterare perché la $f(c)$ trovata alla 41^{a} iterazione aveva modulo inferiore al limite minimo da noi imposto e ci ha fornito come risultato la c calcolata alla 41^{a} iterazione, c che risulta essere una buona approssimazione dello zero reale.
Se invece scegliamo come intervallo

- $[a,b]=[0.5,1]$

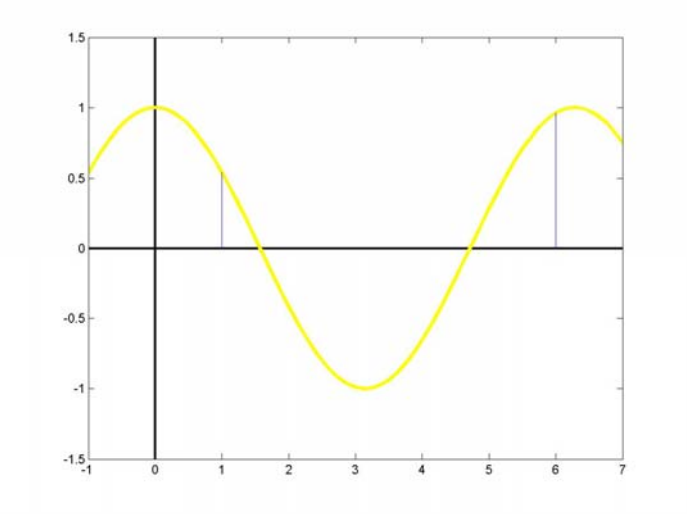
otteniamo che, dato che sia $f(a)$ che $f(b)$ sono positive, il programma abortisce e lancia il messaggio di errore da noi impostato.
Il grafico relativo è riportato di seguito.



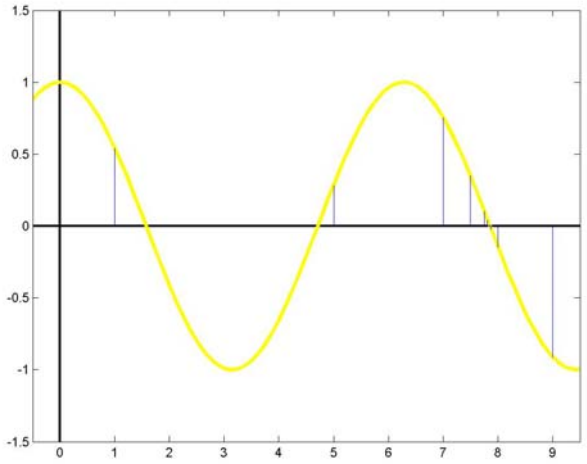
Avremmo lo stesso risultato scegliendo per esempio come intervallo

- $[a,b]=[1,6]$

Il grafico è il seguente.



Per concludere mostriamo un caso in cui il nostro algoritmo fallisce i quanto, dato un intervallo, non riesce ad individuare la presenza di più zeri all'interno di tale intervallo e quindi va a calcolarne uno solo perdendo gli altri.



Nell'immagine qui sopra è visualizzato il caso in cui abbiamo scelto come intervallo

- $[a,b]=[1,3]$

Come si vede dal grafico l'algoritmo converge allo zero che si ha in $x=\frac{5}{2}\pi$ dandone una buona approssimazione in 46 iterazioni. Però non riesce ad individuate gli altri due zeri presenti nell'intervallo di partenza.

Possiamo quindi concludere che il metodo di bisezione risulta essere un metodo di semplice implementazione e abbastanza potente che però è anche fortemente condizionato dalla scelta dell'intervallo iniziale e che, di conseguenza, può fornirci soluzioni incomplete o addirittura errate a seconda della funzione che andiamo ad analizzare.

Metodo di Newton

Il metodo di Newton è un metodo di tipo iterativo che, data una $f(x)$ e un punto iniziale x_0, ci consente di giungere dopo un determinato numero di passi ad un'approssimazione dello zero più vicino al punto di partenza.
Per far ciò si procede nel seguente modo:

1. si trova la retta tangente a $f(x)$ per $x = x_0$
2. individuiamo x_1, punto di intersezione tra la retta e l'asse delle ordinate
3. se $f(x_1) \neq 0$ o il numero di iterazioni effettuate non supera il numero massimo da noi imposto allora torna al punto 1 impiegando x_1 al posto di x_0, altrimenti termina ed x_1 è lo zero da noi cercato

Possiamo implementare in Matlab la procedura sopra esplicata nel seguente modo:

```
a = (punto iniziale)
fa  = (funzione calcolata in a);
fa1 = (derivata in a);

while ((N>k) & (norm(fa)> T*eps) & (norm(a) < O))
        a = a - fa/fa1;
        fa  = (funzione in a);
        fa1 = (derivata in a);

        k= k+1;
end
a
```

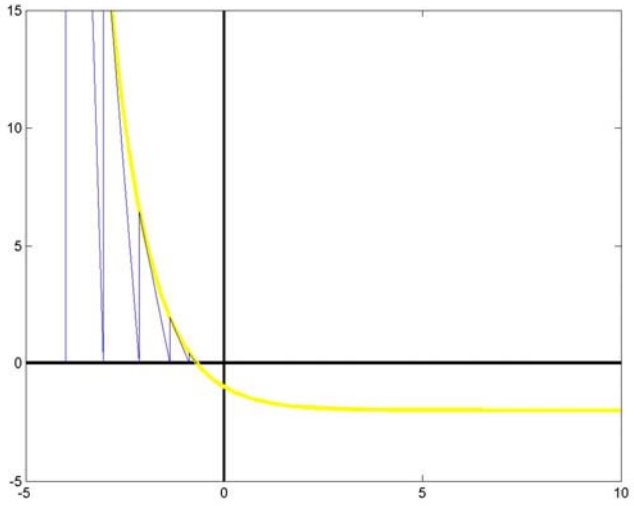
Forniamo ora una spiegazione sul funzionamento. Per prima cosa andiamo a salvare nella variabile a il valore del punto iniziale. Con fa ed fa1 indichiamo due funzioni della variabile a che saranno rispettivamente la funzione di cui noi vogliamo andare a calcolare gli zeri e la sua derivata prima. Dobbiamo scriverle al di fuori del ciclo while altrimenti quest'ultimo non avrà alla prima iterazione parametri sui quali eseguire i controlli ed il programma, di conseguenza ,abortirà.
All'interno del ciclo while andremo a calcolare con l'operazione

```
a = a - fa/fa1;
```

il punto per a per l'iterazione successiva e i relativi valori di fa e fa1. Inoltre andremo a tener conto mediante la variabile k del numero di iterazioni eseguite.
Discutiamo ora del while; al suo interno inseriamo tre vincoli legati da dall'operatore logico and.
Il primo è N>k, e questo vincolo impone che il programma non possa effettuare un numero di iterazioni superiore a N da noi precedentemente impostato. Il secondo vincolo a rigor di logica dovrebbe essere fa!=0, cioè il programma dovrebbe iterare fino ad arrivare a calcolare lo zero

esatto. Noi dobbiamo però tener conto del fatto che il calcolatore opera mediante algebra finita, quindi soggetta ad errori e tolleranze. Di conseguenza è più sensato sostituire tale vincolo con norm(fa)> T*eps, dove T e un parametro da noi impostato a priori e eps indica la precisione di macchina.
L'ultimo vincolo, (norm(a) < O con O impostato da noi a priori, serve nel caso trovassimo nella situazione in cui si abbia uno zero per $x \to \infty$.
Infatti in questo caso, a seconda del punto di inizio il metodo potrebbe convergere proprio a tale zero, causando di conseguenza un overflow.
Infine gli and servono affinché le iterazioni si blocchino nel caso in cui anche uno solo di tali vincoli non sia più rispettato.
Applichiamo adesso il metodo ad un caso pratico.
Come primo caso scegliamo una funzione con uno zero semplice. La funzione è la seguente

$$f(x) = e^{-x} - 2$$



In questa prova abbiamo fissato

- $x_0 = -4$
- $N = 100$
- $O = 10^6$
- $T = 10^6$

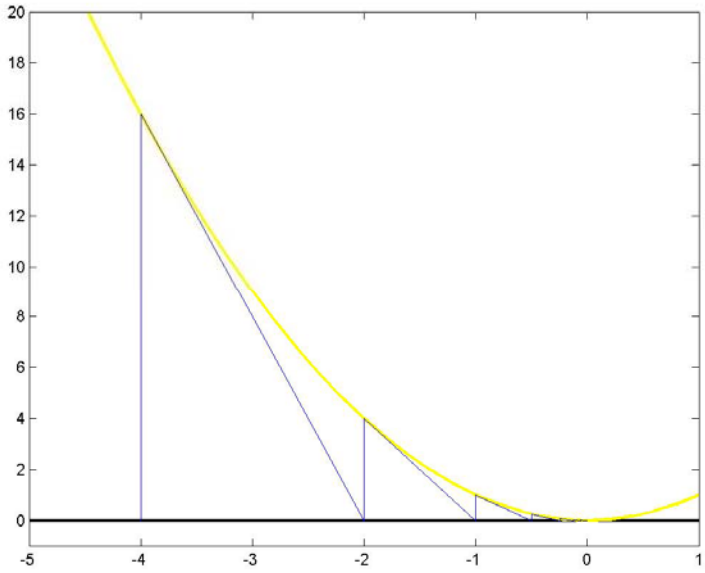
L'algoritmo giunge dopo 8 iterazioni al seguente risultato

$$x_{zero} = -0.6931$$
$$f(x_{zero}) = 0$$

Vediamo ora come si comporta l'algoritmo applicato al una funzione avente uno zero a molteplicità doppia.

La funzione è la seguente

$$f(x) = x^2$$



Ponendo sempre

- $x_0 = -4$
- $N = 100$
- $O = 10^6$
- $T = 10^6$

l'algoritmo giunge dopo 19 iterazioni al seguente risultato

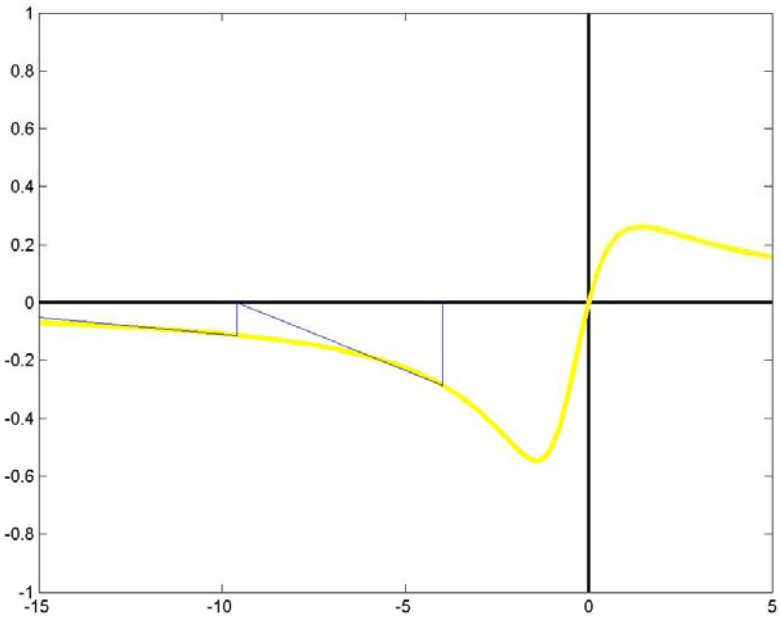
- $x_{zero} = -7.6294 * 10^{-6}$
- $f(x_{zero}) = 5.8208 * 10^{-12}$

Si vede che il metodo anche in questo caso converge molto rapidamente a valori prossimi allo zero. Per aumentare la precisione ci basterà diminuire il valore di T ed aumentare quello di N per permettere un maggior numero di iterazioni quindi una migliore raffinazione della soluzione.
Va comunque notato che in questo caso il metodo ha impiegato per giungere ad una soluzione sufficientemente precisa più del doppio delle iterazioni che aveva compiuto per il caso della funzione con zero a molteplicità singola. Ciò va a confermare quanto ci viene detto dalle teoria, cioè che maggiore è la molteplicità dello zero maggiore sarà il numero di iterazioni necessarie per convergervi.

Proviamo ora cercare con questo metodo gli zeri della seguente unzione:

$$f(x) = \frac{x}{x^2 + x + 2}$$

E' facile vedere che tale funzione ha zeri per $x = 0$ e per $x \to \pm\infty$.
Lasciando gli stessi valori di N,T ed O vediamo cosa succede al variare del punto di inizio.
Per $x_0 = -4$ otteniamo il seguente risultato



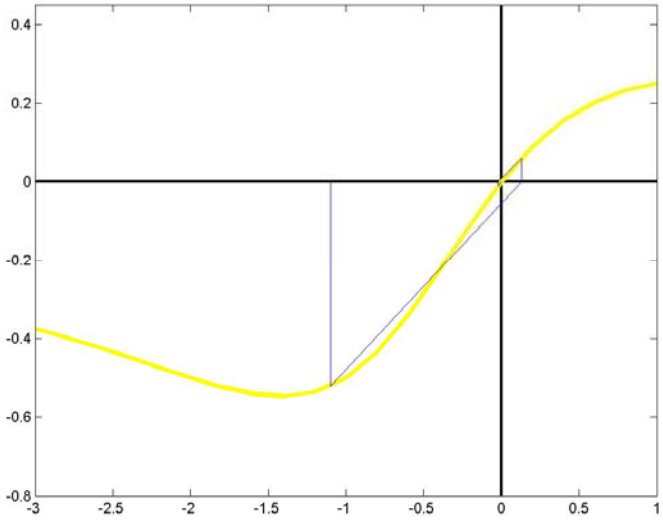
Il metodo si dirige verso lo zero a $-\infty$ terminando dopo sole 18 iterazione per violazione del terzo vincolo del while, poiché siamo arrivati ad ottenere

$$x_{18} = -1.2988 * 10^{6}$$

essendo x_{18} la x calcolata alla diciottesima iterazione.
Se invece poniamo come punto di partenza $x_0 = 1$ otteniamo che il metodo converge alla soluzione $x_{zero} \approx 0$ dopo sole 5 iterazioni. I risultati ottenuti sono i seguenti

- $x_5 = -2.2097 * 10^{-14}$,
- $f(x_5) = -1.1048 * 10^{-14}$

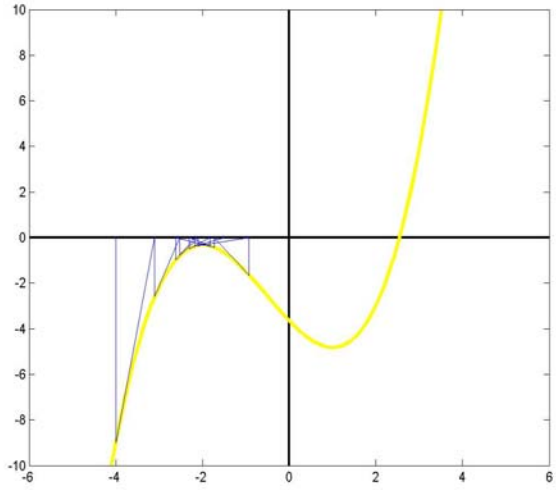
Il grafico è riportato nella pagina seguente.

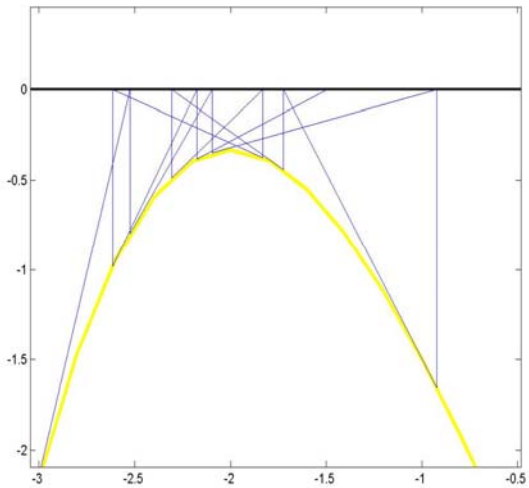


Un altro caso che può essere analizzato è quello in cui il metodo “cada” in un minimo relativo, se si trova nel semispazio $y \geq 0$, e non riesca più ad uscirvi. Se ci troviamo invece nel semispazio $y < 0$ si avrà lo stesso effetto se incappiamo in un massimo relativo. Un esempio di questa situazione può essere dato dalla funzione

$$f(x) = \frac{2x^3 + 3x^2 - 12x - 22}{6}$$

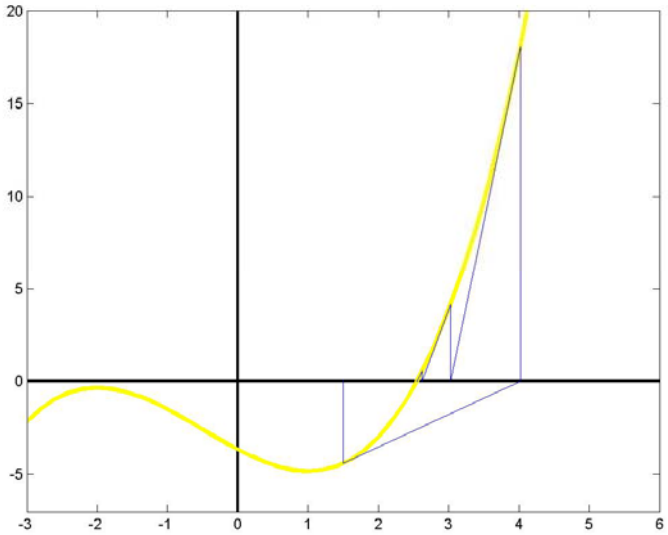
ponendo come punto iniziale $x_0 = -4$ e come numero massimo di iterazioni $N = 10$ otteniamo il risultato mostrato nelle figure seguenti





Si vede che il programma continua ad iterare in prossimità del massimo relativo che si trova in $x = -2$ fino al raggiungimento del numero massimo di iterazioni. Se invece scegliessimo un punto di partenza a destra del minimo relativo che si trova in $x = 1$ saremo sicuri della convergenza del metodo allo zero della funzione.

Il grafico relativo alla situazione in cui poniamo come punto iniziale $x = 1.2$ e $N = 100$ è il seguente



In questo caso giungiamo ad una buon'approssimazione del risultato in sole 6 iterazioni. Otteniamo infatti i seguenti risultati.

- $x_{zero} = 2.5483$
- $f(x_{zero}) = 1.9568 * 10^{-11}$

abbiamo quindi dimostrato che il metodo di Newton, per quanto potente possa essere, presenta una forte dipendenza dalle condizioni iniziali e lo svantaggio computazionale sia di dover conoscere funzione e relativa derivata prima, problema non sempre banale, sia di doverle ricalcolare ad ogni iterazione nel nuovo punto, necessità che può essere onerosa nel caso di funzioni e derivate molto complesse.

Metodi quasi Newton

Metodo delle corde

Il primo metodo che andiamo a considerare è quello delle corde. Esso differisce da quello di Newton solo perché il calcolo sella derivata nel punto viene effettuato solo ogni *n* iterazioni o, in certi casi, esso viene fatto solo alla prima iterazione e poi mantenuto costante per tutte le iterazioni successive.

Per il caso in cui usiamo per tutto il procedimento il valore della derivata calcolato all'inizio una possibile implementazione può essere la seguente.

```
a =  !!punto iniziale
fa = !!f(x) calcolata in a
fa1 = !!derivata di f(x) calcolata in a;

while ((N>k) & (norm(fa) >= T*eps) & (norm(a) < O))
        a = a - fa/fa1;
```

```
        fa = !!f(x) calcolata nella nuova a
        k= k+1;
end
k
fa
a
```

Si vede che l'algoritmo è lo stesso di quello per il metodo di Newton meno che per il fatto che non si ha aggiornamento del valore di fa1 ad ogni iterazione.
Per il caso in cui invece la derivata venga aggiornato ogni n iterazioni l'algoritmo può essere implementato così

```
a =  !!punto iniziale
fa = !!f(x) calcolata in a
fa1 = !!derivata di f(x) calcolata in a;

while ((N>k) & (norm(fa) >= T*eps) & (norm(a) < O))
        for i = 1:10
                a = a - fa/fa1;
                fa = !!f(x) calcolata in a
                k = k+1;
                if((norm(fa) < T) | (k >= N) | (norm(a) < O))
                        break
                end
        end
        fa1 = 2*a;
end
k
fa
a
```

Rispetto all'implementazione del metodo di Newton abbiamo dovuto inserire all'interno del while un'istruzione for. Grazie a tale istruzione il metodo svolge in determinato numero di iterazioni, nel nostro caso dieci, prima di aggiornare il valore della derivata. Inoltre siamo stati costretti inserire, annidata nel for, un'istruzione di tipo if che il compito di bloccare le iterazioni del for mediante l'istruzione break qualora uno dei vincoli imposti nel while venissero violati. Siamo costretti a farlo perché in caso contrario il while potrebbe effettuare il controllo solo dopo ogni ciclo for il che, a seconda dei casi, può risultare inaccettabile.

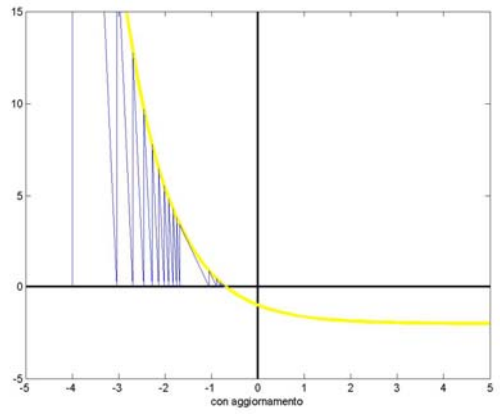
Adesso andremo a testare entrambe le tipologie raffrontandole tra loro e con i risultati ottenuti con il metodo di Newto.Il primo caso è quello della funzione

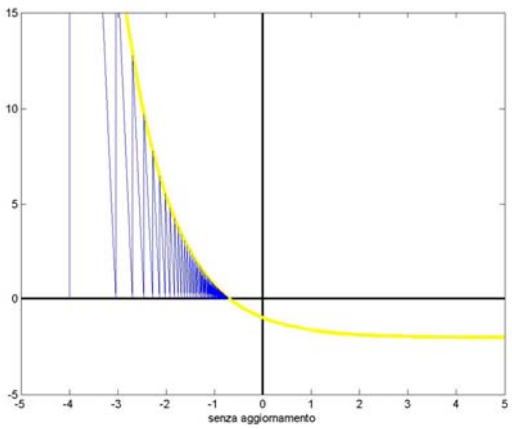
$$f(x) = e^{-x} - 2$$

Avendo posto sempre

- $x_0 = -4$
- $N = 100$
- $T = 10^6$
- $O = 10^6$

Otteniamo i seguenti risultati.





Con aggiornamento

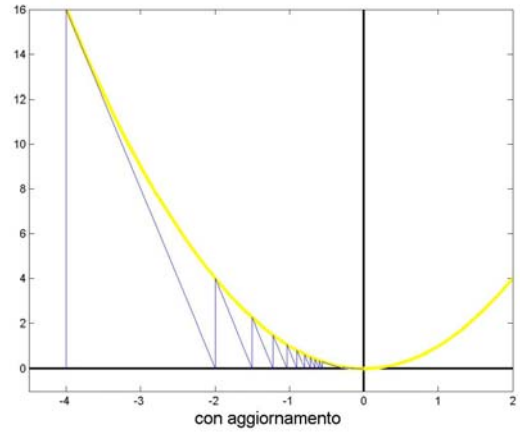
- $k = 24$
- $x_{zero} = -0.6931$
- $f(x_{zero}) = 1.2439 * 10^{-12}$

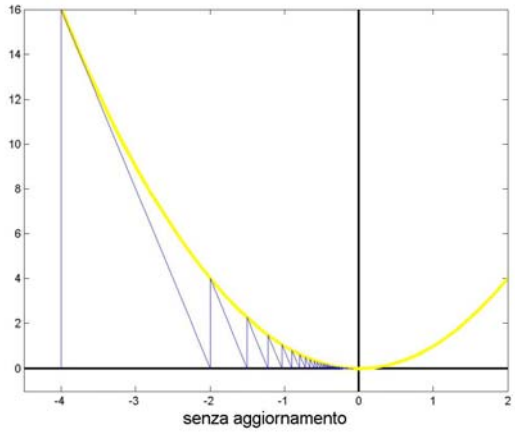
Senza aggiornamento

- $k = 100$
- $x_{zero} = -0.7148$
- $f(x_{zero}) = 0.0439$

Si vede chiaramente che senza un aggiornamento Il metodo impiegherebbe un numero elevatissimo di iterazioni per convergere. Con l'aggiornamento invece arriviamo ad avete una buon'approssimazione dello zero, benché sia più imprecisa di quella ottenuta con il metodo di Newton classico e vi arrivi con più del triplo delle iterazioni.

Analizziamo ora il caso della parabola centrata nell'origine e iterazioni che partono come per il metodo di Newton dal punto $x_0 = -4$.
Come per la funzione precedente otteniamo per i due metodi risultati molto differenti.





Con aggiornamento

- $k = 61$

Senza aggiornamento

- $k = 100$

- $x_{zero} = -1.4364 * 10^{-5}$
- $f(x_{zero}) = 2.0633 * 10^{6}$

- x_{zero} = -0.0752
- $f(x_{zero}) = 0.056$

Si vede anche in questo caso che senza un aggiornamento periodico del coefficiente angolare della retta tangente non si riesce ad avere una buona approssimazione dello zero in un numero ragionevole di iterazioni. Inoltre, riferendoci solo al metodo con aggiornamento, si vede che l'aumentare della molteplicità dello zero porta anche in questo caso ad un'inevitabile aumento del numero di iterazioni necessarie ad avere convergenza.

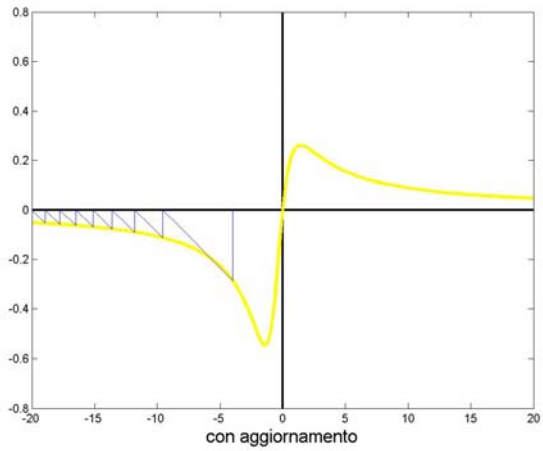
Esaminiamo ora il caso, gia studiato per Newton, in cui applichiamo questi metodi alla seguente funzione:

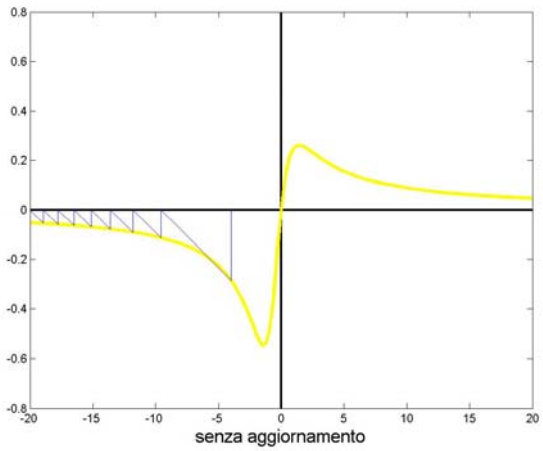
$$f(x) = \frac{x}{x^2 + x + 2}$$

Studiamo il comportamento sempre partendo dai seguenti punti iniziali

- $x_{01} = -4$
- $x_{02} = 1$

Per x_{01} abbiamo il seguente comportamento





Con aggiornamento

- $k = 76$
- $x_{zero} = -1.007 * 10^{6}$
- $f(x_{zero}) = -9.9302 * 10^{-7}$

Senza aggiornamento

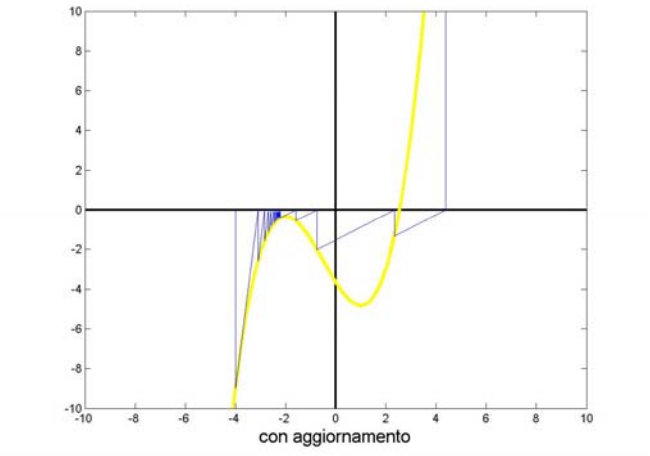
- $k = 100$
- $x_{zero} = -64.1438$
- $f(x_{zero}) = -0.0158$

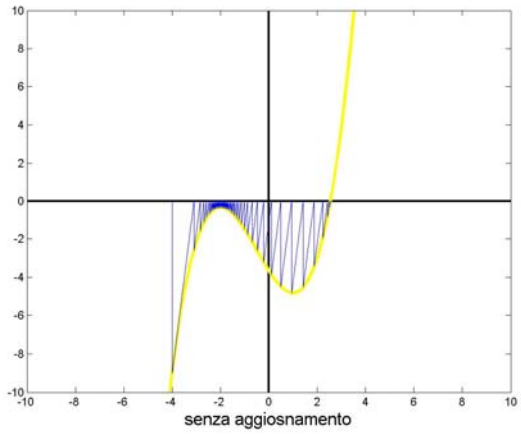
Si vede che il comportamento è sostanzialmente lo stesso che si ha per il metodo di Newton, per quanto il metodo con aggiornamento converga ad infinito in maniera estremamente più veloce di quanto faccia quello senza aggiornamento.

Vediamo ancora cosa succede per la funzione

$$f(x)=\frac{2x^3+3x^2-12x-22}{6}$$

Ponendo sempre come munto iniziale $x_0=-4$ vediamo che il comportamento dei due metodi è molto differente.





Con aggiornamento

- $k=17$
- $x_{zero}=-3.6376*10^{12}$
- $f(x_{zero})=-1.6044*10^{37}$

Senza aggiornamento

- $k=63$
- $x_{zero}=2.5483$
- $f(x_{zero})=-9.5267*10^{-11}$

Il metodo con aggiornamento, aggiornando il coefficiente angolare della tangente solo ogni 10 iterazioni, non riesce a seguire la funzione nei suoi repentini cambi di inclinazione. Di conseguenza, invece di convergere fino ad individuare lo zero, si allontana sempre di più finche le iterazioni non vengono bloccate dal vincolo da noi imposto per intercettare le convergenze ad infinito.
Il metodo senza aggiornamento invece converge allo zero nel giro di 63 iterazioni. Questo però è solo un caso fortuito in quanto se scegliessimo un punto in cui la tangente presenti un coefficiente angolare negativo, o anche uno positivo ma tale da portarci oltre lo zero della funzione prima che i criteri di arresto interrompano le iterazioni, allora il metodo non convergerebbe mai e le iterazioni si bloccherebbero o per raggiungimento del loro numero massimo o per violazione del vincolo sulla convergenza ad infinito da noi imposto.

Metodo regula falsi
Il metodo regula falsi è molto simile al metodo di Newton, con la differenza che al p posto della tangente alla curva. Questo ci impone di fornire all'algoritmo due punti di inizio ma altresì ci libera dalla necessità di conoscere la derivata della funzione.
L'implementazione di tale metodo può essere la seguente.

```
a = !!x0
b = !!x1
fa = !!funzionw calcolata in a
fb = !!funzione calcolata in b
```

```
while((N>k) & (norm(fc) > T*eps) & (c < O))
   if((norm(fb-fa) < 10*T*eps) | ((abs(c/b) > (O/1000) )& (abs(c/a)>(O/1000))))
      break;
 else
 c = (a*fb-b*fa)/(fb-fa)
 fc = !! funzione calcolata in c
 if(a > c)
         b=a
         a=c
         fb=fa
         fa=fc
else(b < c)
         a=b
         b=c
         fa=fb
         fb=fc
 end
k = k+1
end
end
fc
c
```

Quello riportato qui sopra è il nucleo del programma che implementa il metodo delle secanti. Il compito di controllare il numero di iterazioni svolte, il raggiungimento dello zero o la convergenza ad infinito è affidato ad un costrutto di tipo while. Le costanti K ed O sono le stesse definite per il metodo di Newton. All'interno del costrutto while possiamo poi vedere la presenza di un'istruzione if-else: tale istruzione ha il compito di bloccare il programma nel caso raggiungessimo un valore di fb-fa talmente piccolo da rischiare di generare un overflow durante il calcolo della c successiva oppure nel caso in cui ci trovassimo in presenza di una tangente orizzontale.
Nel caso ciò non avvenisse viene calcolato il nuovo punto, indicato con c, ed il corrispondente valore della funzione. Dopo tale calcolo, mediante l'istruzione if-elseif-else adiamo a preparare i nuovi punti a e b per l'iterazione successiva, discernendo tra il caso in cui c si trovi a sinistra di a oppure a destra di b. Una volta ultimate le iterazioni ci vengono restituiti in uscita il numero di iterazioni, lo zero trovato ed il valore della funzione in quello zero, in modo da trarre le opportune considerazioni di risultati ottenuti.

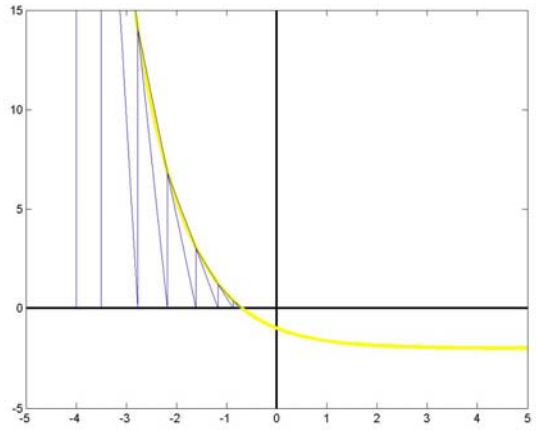
Testiamo ora il tale algoritmo.
Iniziamo sempre con la funzione

$$f(x) = e^{-x} - 2$$

Poniamo

- $N = 100$
- $O = 10^6$
- $T = 10^3$

Scegliamo come punti iniziali –4 e –3.5
Riportiamo i risultati ottenuti nella pagina seguente



Per –4 , -3.5

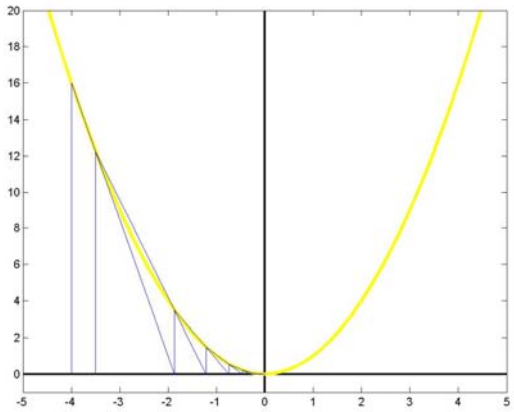
- $k = 11$
- $x_{zero} = -0.6931$
- $f(x_{zero}) = 0$

Si vede chiaramente che il metodo risulta essere solo lievemente più lento di quello di Newton, altresì impiega meno della metà delle iterazioni necessarie al metodo delle corde con aggiornamento per giungere a soluzione.

Testiamo ora il metodo regula falsi sulla funzione

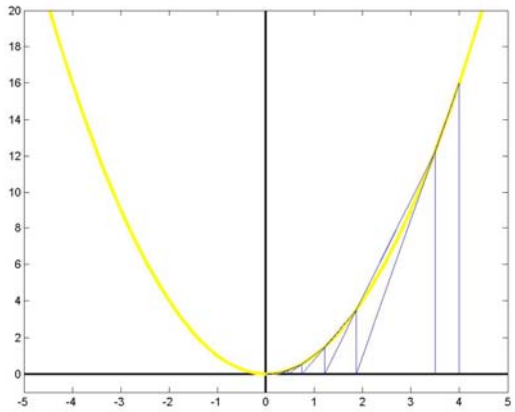
$$f(x) = x^2$$

impiegando come punti di inizio prima –4 e –3.5 e poi 3.5 e 4.
Il risultato è riportato qui sotto



Per –4 , -3.5

- $k = 31$
- $x_{zero} = -1.0428 * 10^{-6}$
- $f(x_{zero}) = 1.0875 * 10^{-12}$



Per 3.5 , 4

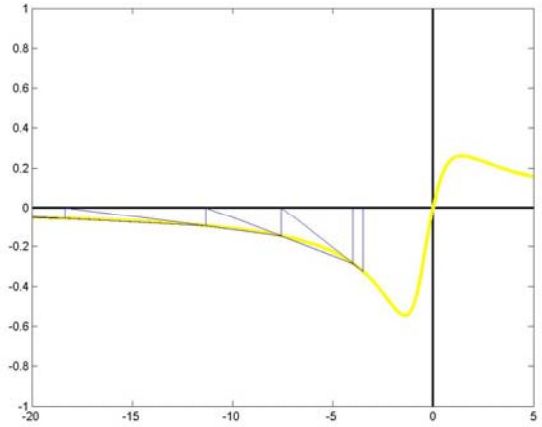
- $k = 31$
- $x_{zero} = 1.0428 * 10^{-6}$
- $f(x_{zero}) = 1.0875 * 10^{-12}$

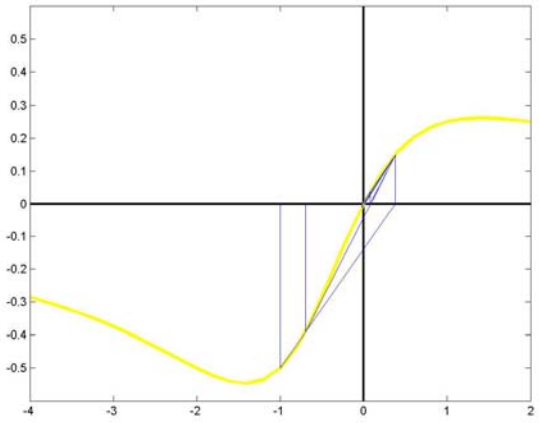
Si vede molto chiaramente che il nostro algoritmo converge in entrambi i casi ad una buona approssimazione della soluzione con un numero di iterazioni non eccessivamente alto, anche qui di poco superiore a quello del metodo di Newton ma inferiore a quello del metodo delle corde con aggiornamento.

Ora, come fatto per gli altri algoritmi, testiamo facciamo delle prove sulla seguente funzione.

$$f(x)=\frac{x}{x^2+x+2}$$

Riportiamo i grafici per diversi punti di inizio.





Per –4 ,-3.5

- $k=26$
- $x_{zeero}=-1.1131*10^{6}$
- $f(x_{zero})=-8.9835*10^{-7}$

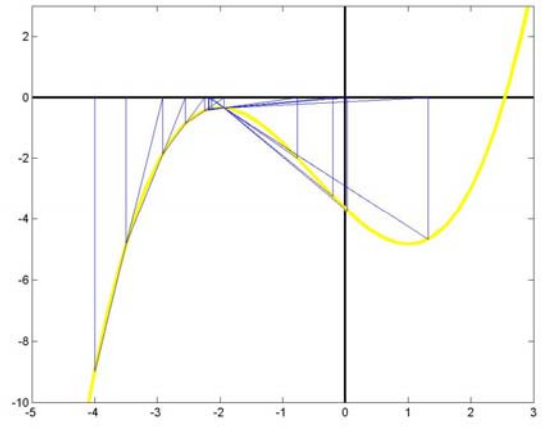
Per –1 , -0.7

- $k=22$
- $x_{zero}=1.5972*10^{-13}$
- $f(x_{zero})=7.9862*10^{-14}$

Come si vede facilmente dalla tabella riportata qui sopra il metodo delle secanti si comporta sostanzialmente come il metodo di Newton e quello delle corde.
Studiamo ancora il comportamento di tale metodo per la funzione

$$f(x)=\frac{2x^3+3x^2-12x-22}{6}$$

Ponendo $N=12$ otteniamo il seguente risultato.



Per –4 , -3.5

- $k=12$
- $x_{zero}=-2.1374$
- $f(x_{zero})=-0.3682$

Si vede che anche in questo caso il metodo delle secanti si comporta sostanzialmente come il metodo di Newton, cioè rimane imprigionato nell'intorno del massimo relativo situato in $x = -2$.

Metodo delle secanti

Un altro metodo interessante da esaminare è quello selle secanti. Tale metodo è pressoché identico al metodo regula falsi, con la differenza però che la scelta dei punti necessari per l'iterazione successiva non avviene secondo un criterio che tiene conto della posizione del punto *c* rispetto ad *a* e *b* ma, invece, si ci limita a porre *a* uguale a *b* e *b* uguale a *c*. Sotto riportiamo l'implementazione di tale metodo.

```
a = !!x0
b = !!x1
fa = !!funzionw calcolata in a
fb = !!funzione calcolata in b

while((N>k) & (norm(fc) > T*eps) & (c < O))
   if((norm(fb-fa) < 10*T*eps) | ((abs(c/b) > (O/1000) )& (abs(c/a)>(O/1000))))
      break;
 else
 c = (a*fb-b*fa)/(fb-fa)
 fc = !! funzione calcolata in c
a=b
b=c
fa=fb
fb=fc

k = k+1
end
end
fc
c
```

La particolarità del metodo delle secanti risiede nel fatto che, a differenza di tutti i metodi fino ad adesso esaminati, il suo ordine non è 1 ma vale

$$p = \frac{1+\sqrt{5}}{2} = 1.618$$

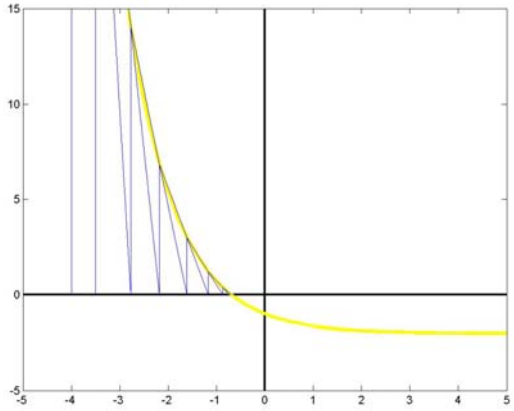
noto come rapporto aureo. Ciò significa che il metodo delle secanti convergerà più velocemente a soluzione rispetto a gli altri metodi studiati.
Esaminiamo ora il comportamento dei questo metodo usando gli stessi parametri impiegati per il metodo regula falsi, cioè imponendo

- $N = 100$
- $O = 10^6$
- $T = 10^3$

La prima funzione è

$$f(x) = e^{-x} - 2$$

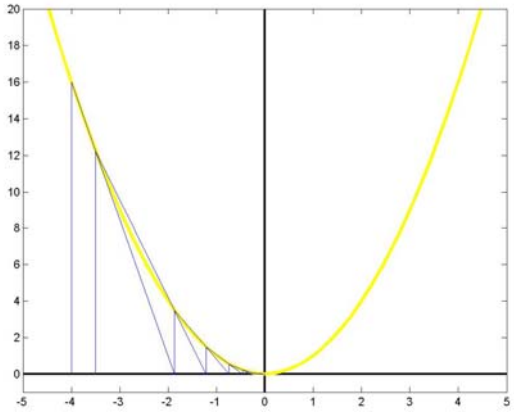
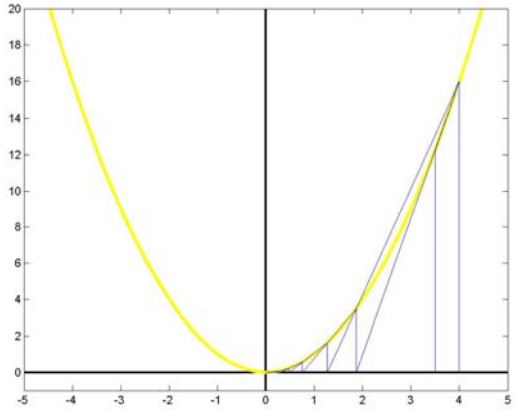
Riportiamo sotto i risultati ottenuti

	Per –4 , -3.5 • $k = 11$ • $x_{zero} = -0.6931$ • $f(x_{zero}) = 0$

Si vede che il comportamento è l0 stesso riscontrato per il metodo regula falsi.
Procediamo oltre. Ora testiamo il metodo sulla parabola con centro nell'origine

$$f(x) = x^2$$

impiegando sempre come punti di inizio prima –4 e –3.5 e poi 3.5 e 4.

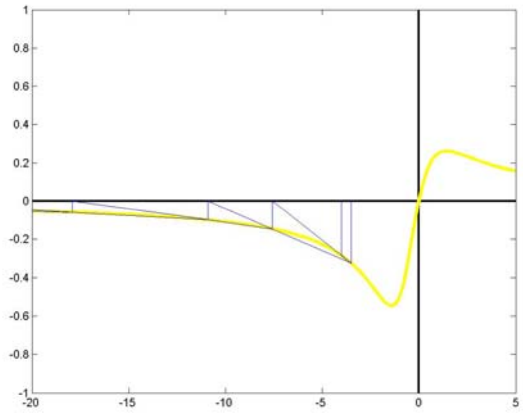
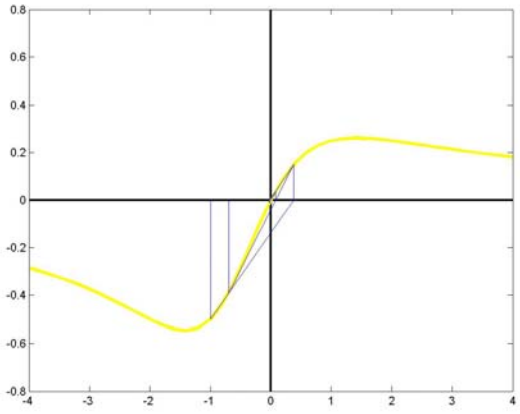
	
Per –4 , -3.5 • $k = 31$ • $x_{zero} = -1.0428 * 10^{-6}$ • $f(x_{zero}) = 1.0875 * 10^{-12}$	Per –4 , -3.5 • $k = 31$ • $x_{zero} = -1.0762 * 10^{-6}$ • $f(x_{zero}) = 1.1581 * 10^{-12}$

In questa prova si riscontra una discrepanza coi i risultati ottenuti con il metodo regula falsi. Infatti, mentre per i punti iniziali –4, -3.5 i due metodi hanno lo stesso comportamento, per i punti iniziali 3.5, 4 il metodo delle secanti impiega lo stesso numero di iterazioni del regula falsi gungendo però ad una soluzione meno accurata.

Continuando le prove come fatto per i metodi precedenti otteniamo per la funzione

$$f(x)=\frac{x}{x^2+x+2}$$

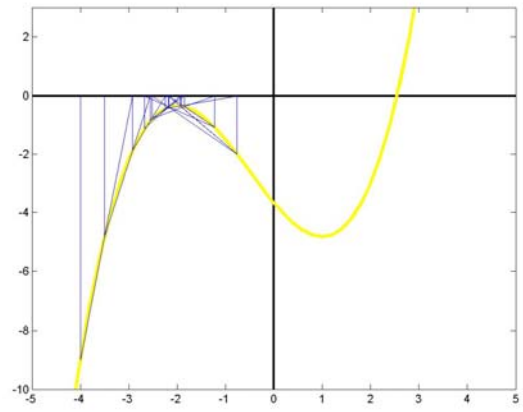
i seguenti risultati

	
Per –4 ,-3.5 • $k=26$ • $x_{zeero}=$ -1.0819*10^{6} • $f(x_{zero})=$ -9.2427*10^{-7}	Per –1 , -0.7 • $k=7$ • $x_{zero}=2.2193*10^{-14}$ • $f(x_{zero})=1.1097*10^{-14}$

In questa prova si vede che lo scostamento tra il comportamento del metodo regula falsi e quello delle secanti e molto marcato. Mentre il comportamento per i punti iniziali –4, -3.5 risulta essere praticamente lo stesso per i punti iniziali –1, -0.7 il metodo delle secanti giunge ad una soluzione più accurata di quella raggiunta dal regula falsi e lo fa impiegando meno di un terzo delle iterazioni. Questo a dimostrazione che l'ordine di convergenza del metodo delle secanti è superiore di quello del regula falsi e di tutti i metodi precedentemente esaminati.
Come ultimo caso di analisi abbiamo la funzione

$$f(x)=\frac{2x^3+3x^2-12x-22}{6}$$

Avendo posto $N=12$ otteniamo il seguente risultato.



Per –4 , -3.5

- $k=12$
- $x_{zero}=$ -1.9067
- $f(x_{zero})=$ -0.3461

Come quasi tutti gli altri metodi il metodo delle secanti rimane imprigionato nell'intorno del massimo relativo situato in $x = -2$ senza riuscire più ad uscirne.

Riassunto

Ora che abbiamo esaminato tutti i comportamenti dei metodi di bisezione, Newton e quasi Newton possiamo concludere la trattazione di questa parte fornendo una tabella nella quale riportiamo i risultati ottenuti dall'applicazione dei metodi sopra descritti alla funzione in maniera da fornire la possibilità di un rapido confronto tra le prestazioni dei singoli metodi.

Ponendo

- $N = 100$
- $O = 10^6$
- $T = 10^6$

Eseguiamo la prima prova sulla funzione

$$y = e^{-0.5*x} - 8$$

che presenta una radice singola in $x = 0.44629$. Nella tabella seguente sono riportati i risultati ottenuti

Metodo	Punti di inizio	x_{zero}	$f(x_{zero})$	K
Bisezione	$[-10,-5]$	-4.1589	$1.8689*10^{-11}$	36
Newton	-4	-4.1589	$1.1006*10^{-10}$	3
Corde con aggiornamento	-4	-4.1589	$1.1006*10^{-10}$	3
Corde senza aggiornamento	-4	-4.1589	$5.7733*10^{-11}$	9
Regula Falsi	-4,-3.5	-4.1589	$-1.9122*10^{-11}$	8
Secanti	-4,-3.5	-4.1589	$-1.3980*10^{-12}$	5

Per dare un'idea di come rallentano i metodi all'aumentare della molteplicità degli zeri riportiamo anche i risultati ottenuti per la funzione

$$f(x) = x^3$$

Metodo	Punti di inizio	x_{zero}	$f(x_{zero})$	K
Bisezione	$[-4,-1.5]$	$1.2207*10^{-4}$	$1.819*10^{-12}$	12
Newton	-4	$-5.3463*10^{-4}$	$-1.5281*10^{-10}$	22
Corde con aggiornamento	-4	$-5.3463*10^{-4}$	$-1.5281*10^{-10}$	22
Corde senza aggiornamento	-4	-0.4779	-0.1092	100
Regula Falsi	-4,-3.5	$-9.6127*10^{-4}$	$-8.8935*10^{-10}$	29
Secanti	-4,-3.5	$-9.6127*10^{-4}$	$-8.8935*10^{-10}$	29

Metodi Multidimensionali

I metodi multidimensionali sono dei metodi che ci permettono di individuare soluzioni reali di sistemi di *n* equazioni non lineari in *n* incognite. Tale risultato viene ottenuto mediante procedimenti di tipo iterativo. Il primo metodo che andiamo ad analizzare è il metodo di Newton multidimensionale.

Metodi risolutivi

Il metodo di Newton multidimensionale non è nient'altro che l'estensione di quello visto per il caso monodimensionale. Dato il nostro sistema di equazioni noi possiamo indicarlo nel seguente modo

$$F(X)=0$$

dove $X=(x_1,x_2,\cdots,x_n)^T$ e $F:\Re^n\to\Re^n$ è la funzione vettoriale

$$F=\begin{bmatrix} f_1(x_1,x_2,\cdots,x_n) \\ f_2(x_1,x_2,\cdots,x_n) \\ \vdots \\ f_n(x_1,x_2,\cdots,x_n) \end{bmatrix}$$

Quello che noi andiamo a fare è approssimare $F(X)$ con la sua serie di Taylor troncata al termine del prim'ordine. La serie è la seguente

$$F(X)\sim F(X^{(k)})+F'(X^{(k)})(X-X^{(k)})$$

dove $F'(X^{(k)})$ è lo Jacobiano calcolato nel punto $X^{(k)}$.
Come per il metodo di Newton monodimensionale il calcolo dell'iterata successiva i effettua nel seguente modo

$$X^{(k+1)}=X^{(k)}-(F'(X^{(k)}))^{-1}F(X^{(k)})$$

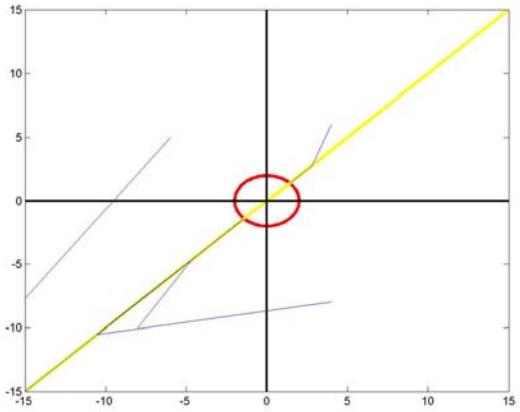
come gia fatto in precedenza forniamo un'implementazione in Matlab di tale algoritmo.

```
x  = !!punto iniziale
x2 = !! posto diverso da x serve per il confronto
     !!nel criterio di arresto
while((N>k)&(norm(n-n2)> t*norm(n2)))
     x2 = x;
     F=!!funzione F(X)
     F1 = !!Jacobiano di F(x)
     x = x - F1\F
     k = k + 1
end
x
```

Si vede chiaramente che lo schema di funzionamento è sostanzialmente lo stesso di quello del caso monodimensionale, con l'unica differenza che al posto di variabili abbiamo vettori e che la matrice Jacobiana sostituisce la derivata prima. Testiamo ora tale metodo sul seguente sistema

$$\begin{cases} y^2 + x^2 - 4 = 0 \\ y - x = 0 \end{cases}$$

Sotto riportiamo il grafico che descrive il comportamento dell'algoritmo per vari punti di inizio.



I risultati che otteniamo sono riportati nella tabella seguente

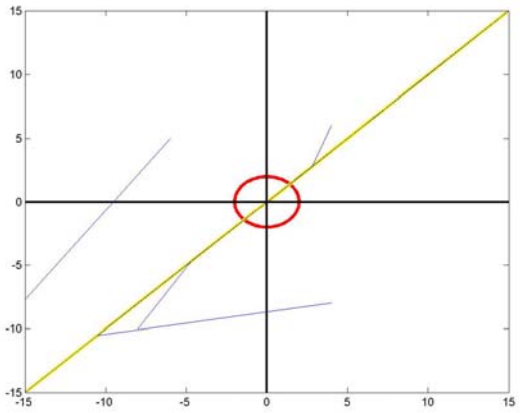
Punti d'inizio	**Risultati**	**Iterazioni**
$[-8,-10]$	$[-1.4142,-1.4142]$	6
$[4,6]$	$[1.4142,1.4142]$	5
$[-6,5]$	$[-1.4142,-1.4142]$	9
$[4,-8]$	$[-1.4142,-1.4142]$	7

Si vede che per tutti i punti di inizio il metodo converge ad una delle due soluzioni in un basso numero di iterazioni. Questo dimostra che il metodo di Newton è molto efficiente anche nel caso multidimensionale.
Come per il caso monodimensionale possiamo pensare di aggiornare lo Jacobiano non ad ogni iterazione ma dopo un certo numero di iterazioni.
Tale metodo può avere la seguente implementazione

```
x  = !!punto iniziale
x2 = !! posto diverso da x serve per il confronto
      !!nel criterio di arresto
```

```
while((N>k)&(norm(x-x2)> t*norm(x2)))
   F1 = !! Jacobiano di F(X)
   for i=1:5
      x2 = x;
      F=!!funzione F(X)
      n = n - F1\F
      k = k + 1
      if((norm(n-n2) < t*norm(n2)))
         break
      end
   end
end
x
```

Come si può vedere lo schema di funzionamento di tale algoritmo è il medesimo di quello del suo corrispettivo monodimensionale.
Testando questo metodo sul medesimo sistema di prima.Ponendo $N = 1000$ e lasciando invariati gli altri parametri otteniamo i seguenti risultati.



Punti d'inizio	**Risultati**	**Iterazioni**
$[-8,-10]$	$[-1.4142,-1.4142]$	12
$[4,6]$	$[1.4142,1.4142]$	9
$[-6,5]$	$[1.4142,1.4142]$	194
$[4,-8]$	$[1.4142,1.4142]$	54

Si vede che anche aggiornando lo Jacobiano ogni tot iterazioni giungiamo agli stessi risultati cui siamo arrivati prima, ma che per farlo impieghiamo un numero d'iterazioni molto maggiore. Infatti, per i punti iniziali $[-6,5]$ e $[4,-8]$, prima che il metodo cominci a convergere si ha un fortissimo

scostamento dalla soluzione in cui poi si va a convergere. Ciò è dovuto al fatto che, aggiornando lo jacobiaono solo ogni *n* iterazioni prima che la direzione in cui il nostro sistema si muove venga corretta il metodo può essersi allontanato anche di molto da quella che è la soluzione che al termine delle iterazioni verrà individuata.

Dal metodo di Newton si possono derivare altri due metodi semplificati noti come metodo di Newton-Jacobi e metodo di Newton-Gauss-Seidel.
I due metodi funzionano esattamente come il metodo di Newton, con l'unica differenza che la matrice Jacobiana ora viene approssimata.
Descrivendo la matrice Jacobiana come

$$F'(X) = D(X) - L(X) - U(X)$$

si ha che il metodo di Newton-Jacobi approssima lo Jacobiano con la sua diagonale, cioè

$$F'(X) \approx D(X)$$

mentre il metodo di Newton-Gauss-Seidel lo approssima con il suo triangolo inferiore

$$F'(X) = D(X) - L(X)$$

Testiamo anche questi due metodi sullo stesso sistema di prima.

Newton-Jacobi

Punti d'inizio	**Risultati**	**Iterazioni**
$[-8,-10]$	$[69.8611,-32.0097]$	500
$[4,6]$	$[-363.5106,-910.1071]$	500
$[-6,5]$	$[64.9977,-29.8758]$	500
$[4,-8]$	$[-135.7803,-317.1314]$	500
$[1.414,1.414]$	$[1.4142,1.4142]$	1
$[1.3,1.3]$	$[-0.6672,-3.7192]$	500

Si vede che il metodo di Newton-Jacobi non riesce a convergere per il nostro sistema ad una delle due soluzioni se non quando il punto iniziale è a ridosso di una soluzione, ma basta un piccolo scostamento perché non riesca più a convergere.

Newton-Gauss-Seidel

Punti d'inizio	**Risultati**	**Iterazioni**
$[-8,-10]$	$[1,1]$	500
$[4,6]$	$[-1,-1]$	500
$[-6,5]$	$[1.6,1.6]$	500
$[4,-8]$	$[-0.3636,-0.3636]$	500
$[1.414,1.414]$	$[1.4142,1.4142]$	1
$[1.3,1.3]$	$[1.3,1.3]$	500

Con il metodo di Newton-Gauss-Seidel otteniamo delle soluzioni che sono solo delle grossolane approssimazioni di quelle che sono le vere soluzioni del sistema. Quindi è facile intuire che più il metodo di Newton viene semplificato più la precisione nell'individuare le eventuali soluzioni viene meno.

Applicazione ad un Problema Reale

Il problema di individuare soluzioni per un sistema di equazioni non lineari mi ha interessato giacché mi è capitato di trovarmi di fronte ad un problema di questo genere durante la realizzazione di una tesina per l'esame di Affidabilità dei Sistemi Elettronici che ho svolto in collaborazione con un mio collega. Scopo della tesina era quello di realizzare delle prove di vita accelerata su dei ganci da bagno in maniera da poterne stimare la vite media se sottoposti a regime di utilizzo normale.
Il tipo durante la sperimentazione utilizzammo il peso fattore di stress per il calcolo della vita media del dispositivo. Considerando come fattore di invecchiamento il peso la vita media del gancio in funzione del peso può essere calcolata utilizzando la seguente legge

$$\mu = A + \frac{B}{P^n}$$

nota come legge di Arrhenius, dove $\mu = \ln(t_m)$, con t_m tempo al quale si ha il guasto del 50% dei dispositivi, P il peso applicato e A, B e n costanti da determinare.La non linearità è data dalla presenza di n ad esponente di P.
Da tre prove di vita accelerata ricavammo tre coppie di valori μ e P per poter costruire un sistema di tre equazioni in tre incognite. Quello che ottenemmo fu il seguente sistema

$$\begin{cases} A + \dfrac{B}{P_1^n} - \mu_1 = 0 \\ A + \dfrac{B}{P_2^n} - \mu_2 = 0 \\ A + \dfrac{B}{P_3^n} - \mu_3 = 0 \end{cases}$$

con $P_1 = 1$ kg, $P_2 = 1.7$ kg, $P_3 = 1.35$ kg, $\mu_1 = 2.1318$, $\mu_2 = 0.3597$ e $\mu_3 = 0.8329$.
Per risolvere tale sistema il mio collega ebbe l'idea di impiegare un metodo abbastanza casereccio che c'era stato insegnato durante il corso di dispositivi elettronici. Tale metodo consiste nello scegliere ad arbitrio il valore di una variabile ed utilizzare tale valore per risolvere il sistema formato dalle prime due equazioni, trovando così un primo valore per le altre due incognite. Fatto ciò si sostituiscono i due valori trovati nella terza equazione in modo da trovare un valore più corretto per la prima variabile da noi imposta. Quindi si ricostituisce il nuovo valore nelle prime due equazioni e si itera il procedimento fino a convergenza. Per far ciò il mio collega sviluppo un programmino in C di cui riporto il listato.

```
•#include <stdio.h>
•#include <math.h>•int main(){
•   double A;
•   double B;
•   double n;
•   double mu1 = 2.1318;
•   double mu2 = 0.3597;
•   double mu3 = 0.8329;
•   double p1 = 1;
•   double p2 = 1.7;
•   double p3 = 1.35;
•   char choice;
•   int nIt = 0;
```

```
    printf("Inserisci n: ");
    scanf("%lf", &n);
    printf("\n\nNuova Iterazione? ");
    scanf("%c", &choice);
    printf("\n");
    fflush(stdin);
    while (choice != 'n'){
        double temp = pow(p2,n);
        B = (mu1 - mu2)/(1-pow(temp, -1));
        A = mu1 - B;
        n = log10(B/(mu3-A))/log10(p3);
        nIt++;
        printf("A = %lf\nB = %lf\nn = %lf\nNumero Iterazioni = %d", A, B, n, nIt);
        printf("\n\nNuova Iterazione? ");
        scanf("%c", &choice);
        fflush(stdin);
    }
    getchar();
    fflush(stdin);
    return 0;
}
```

Tale programmino ci permise di giungere in circa 50 iterazioni alla seguente soluzione:

- $A = -0.170224$
- $B = 2.302224$
- $n = 2.768223$

Mio intento è di applicare a tale sistema i metodi esplicati in precedenza per verificare la correttezza dei risultati ottenuti durante la simulazione impiegando il metodo di sostituzione sopra descritto..
Per prima cosa applichiamo il metodo di Newton e vediamo quali risultati ci da.
I risultati ottenuti a partire da diversi punti iniziali sono i seguenti.

Punti d'inizio $[A, B, n]$	**Risultati** $[A, B, n]$	**Iterazioni**
$[5,5,5]$	$[-0.1702, 2.3022, 2.7682]$	5
$[-25,12,45]$	$[-0.1702, 2.3022, 2.7682]$	2
$[50,100,-65]$	$[-0.1702, 2.3022, 2.7682]$	8
$[-0.1,2,2]$	$[-0.1702, 2.3022, 2.7682]$	5

Si vede che i risultati coincidono per tutti i punti di partenza con i risultati che ottenemmo allora con il nostro metodo.
Sperimentiamo ora se anche con gli altri due metodi otteniamo lo stesso risultato.
Per il metodo quasi Newton i risultati sono i seguenti.

Punti d'inizio $[A, B, n]$	**Risultati** $[A, B, n]$	**Iterazioni**
$[5,5,5]$	$[-0.1694, 2.3026, 2.7715]$	7
$[-25,12,45]$	$[\text{NaN}, \text{NaN}, \text{NaN}]$ matlab individua una divisione per zero	2
$[50,100,-65]$	$[2.1332, -65]$ errore, matrice $F'(X)$ malscalata	2
$[-0.1,2,2]$	$[-0.1727, 2.3059, 2.7715]$	2

Per il metodo quasi Newton si vede che a seconda del punto d'inizio il metodo può convergere oppure no. Quindi la perdita di precisione dovuta alla semplificazione del metodo ci portati ad avere una dipendenza dalle condizioni iniziali abbastanza marcata.
Per concludere vediamo come si comportano anche i metodi di Newton-Jacobi e Newton-Gauss-Seidel.

Newton-Jacobi

Punti d'inizio $[A, B, n]$	**Risultati** $[A, B, n]$	**Iterazioni**
$[5,5,5]$	$10^{29} * [2.8165, \text{inf}, \text{NaN}]$ matrice numericamente singolare	5
$[-25,12,45]$	$10^{11} * [-5.9477, \text{NaN}, \text{NaN}]$ matrice numericamente singolare	2
$[50,100,-65]$	$[2.1332, -\text{Inf}, \text{NaN}]$ matrice numericamente singolare	3
$[-0.1702, 2.30224, 2.768223]$	$[-0.1691, 2.3021, 2.7683]$	1

Newton-Gauss-Seidel

Punti d'inizio $[A, B, n]$	**Risultati** $[A, B, n]$	**Iterazioni**
$[5,5,5]$	$10^{7} * [-1.9231, \text{Inf}, \text{NaN}]$ matrice numericamente singolare	3
$[-25,12,45]$	$10^{11} * [-2.3984, \text{Inf}, \text{NaN}]$ matrice numericamente singolare	2
$[50,100,-65]$	$[2.1332, -\text{Inf}, \text{NaN}]$ matrice numericamente singolare	3
$[-0.1702,2.30224,2.768223]$	$[2.1332, \text{NaN}, \text{NaN}]$ matrice numericamente singolare	23

Si vede che l'adozione di semplificazioni drastiche come quelle introdotte nei metodi di Newton-Jacobi e Newton-Gauss-Seidel rendono tale metodo inutile ai fini della risoluzione di questo particolare problema.
In definitiva possiamo trarre come conclusione che diminuendo la complessità del metodo che impieghiamo riduciamo il carico computazionale ma perdiamo anche in precisione, con l'effetto che possiamo ottenere risultati imprecisi o addirittura completamente sbagliati a seconda del sistema d'equazioni su cui andiamo ad operare.